From Computing to Intelligence

A First-Principles Approach to Artificial Intelligence in Language, Vision, and Audio

Sagar Udasi

MSc Statistics with Data Science and Computational Finance
University of Edinburgh

From Computing to Intelligence

A First-Principles Approach to Artificial Intelligence in Language, Vision, and Audio

ISBN: 9798285270225

Author: Sagar Udasi
Publisher: Amazon Kindle Direct Publishing
First Edition

Printed in the United Kingdom

For information, please contact:
Sagar Udasi
Email: sagar.l.udasi@gmail.com

Contents

Introduction

Who Is This Book For?

What This Book Covers and What It Doesn't

Welcome to **From Computing to Intelligence**.

Sagar Udasi

SECTION 1

FOUNDATIONS OF INTELLIGENCE

Chapter 1

Neural Networks

Humans have always considered themselves intelligent, perhaps too intelligent. For centuries, philosophers, anatomists, and scientists stared into the mirror of the mind, trying to grasp what makes intelligence possible. Naturally, the question arose: *How do we think?* And if we could understand that, *could we build a machine that thinks?* To build an artificial intelligence, we need to take inspiration from nature's most optimized real intelligent machine — the brain!

1.1 The Neuron

In 1873, an Italian scientist named ***Camillo Golgi*** discovered a silver staining technique that revealed something extraordinary: the fine cellular architecture of the brain. In 1890, a Spanish neuroscientist and artist, ***Santiago Ramón y Cajal*** took Golgi's stain and produced breathtaking drawings that showed brain as connected architecture of individual, separate units — not a continuous mesh as many thought. Cajal proposed what we now call the *Neuron Doctrine*: that the brain is made up of individual units, now known as **neurons**, that communicate across small gaps by transmitting chemicals (signals), which we now call **synapses**.

The exact working of the brain was little understood by the 1940s. However, one thing was known: neurons form a network, and based on the signals they receive from other neurons, they decide whether to release a signal themselves, a process called *firing*, or not. Enter ***Warren McCulloch***, a neurophysiologist, and ***Walter Pitts***, a brilliant young logician who had taught himself formal logic as a teenager. In 1943, in a small research lab in Chicago, they published a revolutionary paper: *A Logical Calculus of the Ideas Immanent in Nervous Activity* [2].

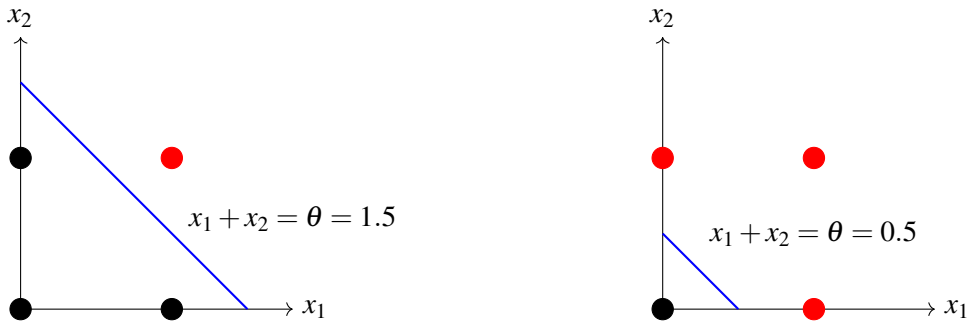
In their paper they proposed that each neuron functions as a binary threshold unit. Neurons receive multiple input signals, aggregate them, and produce an output (i.e., *fire*) only if the aggregated signal crosses a certain threshold. Formally, the **McCulloch-Pitts model** defines a neuron that receives inputs $\mathbf{x} = (x_1,\ x_2,\ \ldots,\ x_n)$, where any input $x_i \in \{0,1\}$. The neuron aggregates the inputs by computing the sum $g(\mathbf{x}) = \sum_{i=1}^{n} x_i$.

The output is then determined by a fixed constant known as the *thresholding parameter*, θ, as

$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta \\ 0 & \text{if } g(\mathbf{x}) < \theta \end{cases}$$

Geometrically, a single McCulloch-Pitts neuron divides the space of input points into two regions by a decision boundary defined by the equation $\sum_{i=1}^{n} x_i - \theta = 0$. For the case of two binary inputs, this line partitions the four possible input points into two groups: those for which $\sum x_i < \theta$ (which produce output 0) and those for which $\sum x_i \geq \theta$ (which produce output 1). Thus, the decision boundary acts as a separator in the input space.

This idea naturally extends to higher dimensions. If the neuron receives three inputs, the line becomes a plane: $\sum_{i=1}^{3} x_i - \theta = 0$. For instance, in the case of the 3-input OR function, the desired plane must separate the point (0,0,0) which should produce output 0, from the other seven binary combinations, which should all produce output 1.



(a) AND Gate (b) OR Gate

Figure 1.1: McCulloch-Pitts neuron implementing logical functions via thresholding

The McCulloch-Pitts neuron provides a foundational abstraction of a biological neuron, but it also raises several important questions. *What if the inputs are not binary but real-valued? Do we always need to manually specify the threshold parameter? Are all inputs equally important, or should some be assigned more weight?* Finally, *can such a neuron model functions that are not linearly separable?*

1.2 The Perceptron

To address these concerns, ***Frank Rosenblatt*** introduced the *Perceptron* in 1958, funded by the U.S. Navy. Rosenblatt's model generalized the McCulloch-Pitts neuron by allowing real-valued inputs, learnable weights, and thresholds.

Formally, the perceptron computes a weighted sum of its inputs and applies a threshold to determine the binary output.

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i - \theta \geq 0 \\ 0 & \text{if } \sum_{i=1}^{n} w_i x_i - \theta < 0 \end{cases}$$

A more commonly used convention rewrites this using a *bias* term.

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i \geq 0 \\ 0 & \text{if } \sum_{i=0}^{n} w_i x_i < 0 \end{cases} \quad \text{where } x_0 = 1 \text{ and } w_0 = -\theta$$

This formulation shows that a perceptron still divides the input space into two halves. Inputs that produce an output of 1 lie on one side of the decision boundary $\sum w_i x_i = 0$, while those producing 0 lie on the other.

In other words, a single perceptron can only model linearly separable functions. However, it differs from the McCulloch-Pitts model in two significant ways: the weights (including the threshold) are learnable from data, and the inputs can be real-valued.

Let us revisit the Boolean OR function. For inputs $x_1, x_2 \in \{0,1\}$, we desire an output $y = x_1 \vee x_2$. The perceptron should satisfy the following inequalities.

$\mathbf{x_1}$	$\mathbf{x_2}$	**OR**	**Inequality**	**Condition**
0	0	0	$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0$	$\Rightarrow w_0 < 0$
1	0	1	$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0$	$\Rightarrow w_1 \geq -w_0$
0	1	1	$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0$	$\Rightarrow w_2 \geq -w_0$
1	1	1	$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0$	$\Rightarrow w_1 + w_2 \geq -w_0$

Table 1.1: Linear inequalities that represent the OR gate using weights w_0, w_1, and w_2

One possible solution satisfying all the above constraints is $w_0 = -1,\ w_1 = 1.1,$ and $w_2 = 1.1$.

Note that this is not a unique solution—many combinations of weights satisfy the inequalities. The key insight is that for linearly separable functions like OR, a perceptron can find suitable weights to model the function accurately. We will later explore how these weights can be learned automatically using the *perceptron learning algorithm* [1].

1.2.1 Perceptron Learning Algorithm

Imagine we want to make a binary decision of whether to watch a movie or not. Suppose we are given a list of m movies, each labeled as either liked (1) or not liked (0) by a user. Each movie is represented by n features, which can be either Boolean or real-valued. We assume the data is linearly separable and aim to train a perceptron to learn this classification rule. In other words, we want the perceptron to find the parameters $\mathbf{w} = [w_0, w_1, \ldots, w_n]$ that define a separating hyperplane.

We now describe the perceptron learning algorithm formally.

Perceptron Learning Algorithm

Let $P \leftarrow$ set of inputs labeled 1
Let $N \leftarrow$ set of inputs labeled 0
Initialize weight vector $\mathbf{w}$ randomly
while not converged **do**
  Pick a random example $\mathbf{x} \in P \cup N$
  if $\mathbf{x} \in P$ and $\mathbf{w}^T\mathbf{x} < 0$ **then**
    $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$
  if $\mathbf{x} \in N$ and $\mathbf{w}^T\mathbf{x} \geq 0$ **then**
    $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$
end while

The algorithm continues updating the weights until all inputs are correctly classified. *But why does this work?*

To understand this, consider two vectors: the weight vector $\mathbf{w} = [w_0, w_1, \ldots, w_n]$ and an input vector $\mathbf{x} = [1, x_1, \ldots, x_n]$. The perceptron computes the dot product

$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{n} w_i x_i$$

The classification rule can then be written as

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} \geq 0 \\ 0 & \text{if } \mathbf{w}^T \mathbf{x} < 0 \end{cases}$$

Geometrically, the equation $\mathbf{w}^T \mathbf{x} = 0$ defines a hyperplane that splits the input space into two halves. Any point $\mathbf{x}$ lying on this hyperplane satisfies $\mathbf{w}^T \mathbf{x} = 0$. The weight vector $\mathbf{w}$ is orthogonal to this hyperplane, because the angle α between $\mathbf{w}$ and any vector $\mathbf{x}$ on the hyperplane is 90°, which implies $\cos \alpha = 0$ and hence $\mathbf{w}^T \mathbf{x} = 0$.

Now consider a point $\mathbf{x} \in P$ (positive class) such that $\mathbf{w}^T \mathbf{x} < 0$. This implies that the angle between $\mathbf{w}$ and $\mathbf{x}$ is greater than 90°, i.e., $\mathbf{x}$ lies in the wrong half-space. We wish to reduce this angle so that $\mathbf{x}$ lies in the correct half-space. Updating $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$ has the following effect

$$\mathbf{w}_{\text{new}}^T \mathbf{x} = (\mathbf{w} + \mathbf{x})^T \mathbf{x} = \mathbf{w}^T \mathbf{x} + \mathbf{x}^T \mathbf{x}$$

Since $\mathbf{x}^T \mathbf{x} > 0$, we have $\mathbf{w}_{\text{new}}^T \mathbf{x} > \mathbf{w}^T \mathbf{x}$, which implies $\cos(\alpha_{\text{new}}) > \cos(\alpha)$ and therefore $\alpha_{\text{new}} < \alpha$. This update moves the decision boundary in the desired direction, reducing misclassification.

Similarly, for a point $\mathbf{x} \in N$ (negative class) such that $\mathbf{w}^T \mathbf{x} \geq 0$, the vector $\mathbf{x}$ lies in the wrong half-space. The angle between $\mathbf{w}$ and $\mathbf{x}$ is less than 90°. Updating $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$ yields

$$\mathbf{w}_{\text{new}}^T \mathbf{x} = (\mathbf{w} - \mathbf{x})^T \mathbf{x} = \mathbf{w}^T \mathbf{x} - \mathbf{x}^T \mathbf{x}$$

Since $\mathbf{x}^T \mathbf{x} > 0$, this results in $\mathbf{w}_{\text{new}}^T \mathbf{x} < \mathbf{w}^T \mathbf{x}$, i.e., $\cos(\alpha_{\text{new}}) < \cos(\alpha)$ and thus $\alpha_{\text{new}} > \alpha$. The update moves the vector $\mathbf{w}$ further from $\mathbf{x}$, pushing the decision boundary away from misclassified negative points.

This geometric intuition explains why the perceptron algorithm always converges for linearly separable data. At each step, it reduces the number of misclassified points by rotating the decision boundary in the correct direction.

Definition 1.1. *Two sets P and N of points in an n-dimensional space are called absolutely linearly separable if there exist $n+1$ real numbers $w_0, w_1, \ldots, w_n$ such that every point $(x_1, x_2, \ldots, x_n) \in P$ satisfies $\sum_{i=1}^{n} w_i x_i > w_0$ and every point $(x_1, x_2, \ldots, x_n) \in N$ satisfies $\sum_{i=1}^{n} w_i x_i < w_0$.*

Theorem 1.1. *If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector $\mathbf{w}_t$ only a finite number of times. In other words, if the vectors in P and N are tested cyclically, a weight vector $\mathbf{w}_t$ is found after a finite number of steps t which separates the two sets.*

Proof. If $\mathbf{x} \in N$, then $-\mathbf{x} \in P'$, where $P' = P \cup \{-\mathbf{x} : \mathbf{x} \in N\}$. This is because if $\mathbf{w}^T \mathbf{x} < 0$, then $\mathbf{w}^T (-\mathbf{x}) > 0$.

Hence, we can consider a unified set P' where every element $\mathbf{p} \in P'$ satisfies $\mathbf{w}^T\mathbf{p} \geq 0$. Without loss of generality, normalize all vectors $\mathbf{p}$ so that $\|\mathbf{p}\| = 1$. This normalization does not affect the separating condition since $\mathbf{w}^T\mathbf{p}/\|\mathbf{p}\| \geq 0$ implies $\mathbf{w}^T\mathbf{p} \geq 0$.

Let $\mathbf{w}^*$ be the normalized ideal weight vector such that $\mathbf{w}^{*T}\mathbf{p} > 0$ for all $\mathbf{p} \in P'$. Define $\delta = \min_{\mathbf{p}\in P'} \mathbf{w}^{*T}\mathbf{p} > 0$.

Now, suppose at time t we inspect point $\mathbf{p}_i \in P'$ and find $\mathbf{w}_t^T\mathbf{p}_i \leq 0$. A correction is made:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{p}_i.$$

Let β be the angle between $\mathbf{w}_{t+1}$ and $\mathbf{w}^*$. Then,

$$\cos\beta = \frac{\mathbf{w}^* \cdot \mathbf{w}_{t+1}}{\|\mathbf{w}_{t+1}\|} = \frac{\mathbf{w}^* \cdot (\mathbf{w}_t + \mathbf{p}_i)}{\|\mathbf{w}_{t+1}\|} = \frac{\mathbf{w}^* \cdot \mathbf{w}_t + \mathbf{w}^* \cdot \mathbf{p}_i}{\|\mathbf{w}_{t+1}\|}.$$

Since $\mathbf{w}^* \cdot \mathbf{p}_i \geq \delta$, we obtain

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_t + \delta.$$

By induction, after k corrections, we have

$$\mathbf{w}^* \cdot \mathbf{w}_t \geq \mathbf{w}^* \cdot \mathbf{w}_0 + k\delta.$$

Now consider the squared norm of $\mathbf{w}_{t+1}$:

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t + \mathbf{p}_i\|^2 = \|\mathbf{w}_t\|^2 + 2\mathbf{w}_t \cdot \mathbf{p}_i + \|\mathbf{p}_i\|^2.$$

Since $\mathbf{w}_t \cdot \mathbf{p}_i \leq 0$ and $\|\mathbf{p}_i\| = 1$, we get

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + 1.$$

Inductively,

$$\|\mathbf{w}_t\|^2 \leq \|\mathbf{w}_0\|^2 + k.$$

Putting everything together,

$$\cos\beta = \frac{\mathbf{w}^* \cdot \mathbf{w}_t}{\|\mathbf{w}_t\|} \geq \frac{\mathbf{w}^* \cdot \mathbf{w}_0 + k\delta}{\sqrt{\|\mathbf{w}_0\|^2 + k}}.$$

As k increases, the numerator grows linearly, but the denominator grows as $\sqrt{k}$, so $\cos\beta$ can grow unbounded. However, since $\cos\beta \leq 1$, the number of corrections k must be bounded. Therefore, the perceptron algorithm must converge after a finite number of updates.

Let's have a look back at the questions raised after McCulloch-Pitts model.

What if the inputs are not binary but real-valued? The perceptron works directly with real-valued inputs, using the sign of the weighted sum $\mathbf{w}^\top\mathbf{x}$ to make decisions.

Do we always need to manually specify the threshold parameter? No, the threshold can be learned as a bias term w_0 by appending a constant $x_0 = 1$ to the input vector.

Are all inputs equally important, or should some be assigned more weight? Inputs can have different importance, reflected in the learned weights $\mathbf{w}$, which adjust based on the training data.

Can such a neuron model functions that are not linearly separable? No, a single perceptron can only separate linearly separable data. We will see soon how to handle this.

This section was developed in 1969 by two prominent MIT scientists, ***Marvin Minsky*** and ***Seymour Papert***. In their book, *Perceptrons*, they critically analyzed the limitations of Rosenblatt's model. They mathematically proved the above mentioned *perceptron learning algorithm* and showed that a single perceptron could not solve non-linearly separable problems, such as the simple XOR function.

Their critique wasn't wrong but the fallout was dramatic. Funding dried up. Neural networks were declared a dead end. For nearly a decade, research interest in neural models collapsed.

1.2.2 Layer of Perceptrons

How many boolean functions can you design from n inputs? 2^{2^n}. Some of them are not linear separable. There is no general formula that tells us that. For 2 input example, there are 16 possible boolean functions, of which XOR and !XOR are not linearly separable.

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 1.2: Functions f_1 to f_{16} for input combinations of x_1 and x_2

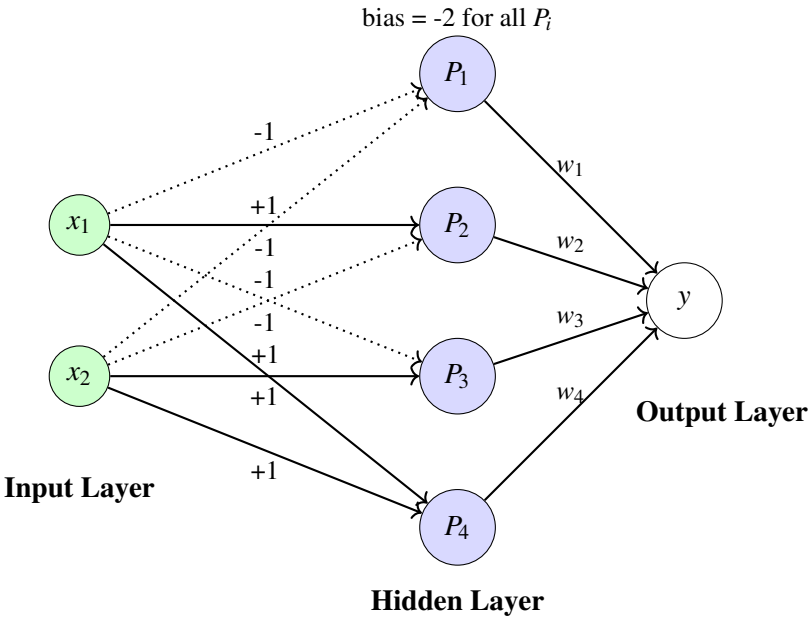


Figure 1.2: Two Input Network with Four Hidden Perceptrons

Two binary inputs, x_1 and x_2, are passed to a hidden layer with four perceptrons P_1 to P_4. Each perceptron in the hidden layer computes a distinct linear combination of the inputs with weights ± 1, effectively separating different input regions. All hidden units use a fixed bias of -2. The outputs from the hidden layer are linearly combined in the output layer using learnable weights w_1, w_2, w_3, w_4 to produce the final output y.

This network can implement *any boolean function*, whether linearly separable or not. The key idea lies in the design of the hidden layer. Each of the four perceptrons is constructed to *fire (output 1) for exactly one of the four possible input combinations* of (x_1, x_2), and *only* for that combination.

Perceptron	**Input that Activates It**
P_1	$(0,0)$
P_2	$(0,1)$
P_3	$(1,0)$
P_4	$(1,1)$

Table 1.3: Unique input activation for each perceptron

Because each hidden unit uniquely represents one input, the output neuron can now learn the desired boolean function by *assigning appropriate weights* w_1, w_2, w_3, w_4 to these hidden activations. For instance, to implement XOR, we can simply set the weights such that the output neuron fires for $(0,1)$ and $(1,0)$, and stays off for $(0,0)$ and $(1,1)$. This translates to the following conditions.

$$w_1 < w_0, \quad w_2 \geq w_0, \quad w_3 \geq w_0, \quad w_4 < w_0$$

There are *no contradictions*. Thus, by adjusting the output weights accordingly, this single network architecture can represent *all 16 possible boolean functions*.

Theorem 1.2. *Any boolean function of n inputs can be represented exactly by a network of perceptrons containing one hidden layer with* 2^n *perceptrons and one output layer containing a single perceptron [1].*

Proof. A boolean function $f : \{0,1\}^n \to \{0,1\}$ is defined on 2^n possible input vectors.

Construct a hidden layer with 2^n perceptrons, each configured to activate only for one unique input vector by appropriate weights and bias.

The output perceptron then sums the hidden layer outputs with weights chosen to output 1 exactly for inputs where f is 1, and 0 otherwise.

Thus, the network exactly represents any boolean function. □

> **Note:** A network with $2^n + 1$ perceptrons is not necessary but sufficient. For example, we have seen how to represent the OR function (and similarly AND function) with just one perceptron.

> **Catch:** As n increases, the number of perceptrons in the hidden layer increases exponentially.

1.3 The Sigmoid Neuron

The perceptron's thresholding logic is quite harsh. Consider a simple example where we decide whether to like or dislike a movie based on a single input: *the critic's rating* x_1, which ranges from 0 to 1. Suppose the threshold is set at 0.5, with weights $w_0 = -0.5$ and $w_1 = 1$. For a movie with $x_1 = 0.51$, the perceptron outputs *like,* while for $x_1 = 0.49$, it outputs *dislike.* This sudden change in decision seems strict and abrupt.

This behavior is not due to the specific problem or chosen weights. Instead, it is inherent to the perceptron function, which acts as a step function. The output switches sharply from 0 to 1 when the weighted sum $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $-w_0$.

In real-world cases, we usually want a smoother decision function that changes gradually from 0 to 1. This motivates the introduction of sigmoid neurons, where the output function is continuous and smooth.

One common sigmoid function is the logistic function, defined as

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$

Here, the output y does not jump suddenly but transitions smoothly around the threshold $-w_0$.

Moreover, the output y is no longer binary; it takes values between 0 and 1. This output can be interpreted as a probability. So instead of a hard like/dislike decision, we get the probability of liking the movie.

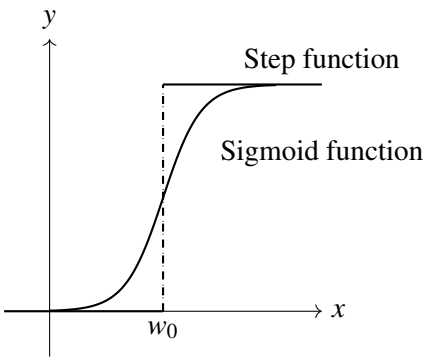


Figure 1.3: Perceptron and Sigmoid Neuron Output Characteristics.

1.3.1 A Typical Supervised Machine Learning Setup

We know that a typical supervised machine learning setup has the following components.

- **Data:** A dataset of n examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, where $\mathbf{x}_i$ are inputs and y_i are corresponding outputs.
- **Model:** An approximation of the relation between input $\mathbf{x}$ and output y. For example,

  $$\hat{y} = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}, \quad \hat{y} = \mathbf{w}^\top \mathbf{x}, \quad \hat{y} = \mathbf{x}^\top \mathbf{W} \mathbf{x}$$

  or any other function mapping inputs to outputs.

- **Parameters:** The model depends on parameters $\mathbf{w}$ (and possibly bias b) that need to be learned from the data.
- **Learning Algorithm:** A method to adjust parameters $\mathbf{w}$ and b to fit the data. Examples include perceptron learning or gradient descent (we'll see gradient descent in detail in the next section).
- **Objective/Loss Function:** A function that measures the error of the model predictions. The learning algorithm aims to minimize this loss.

Consider data points (x, y) where $x, y \in \mathbb{R}$.

$$\{(1,\ 0.05),\ (3,\ 0.15),\ (5,\ 0.4),\ (6,\ 0.6),\ (7,\ 0.65),\ (9,\ 0.85),\ (10,\ 0.95)\}$$

Our model is

$$\hat{y} = \frac{1}{1 + e^{-(wx+b)}}$$

We choose Mean Squared Error (MSE) as the loss function.

$$\mathscr{L}(w,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

This loss $\mathscr{L}(w,b)$ defines a surface over the two-dimensional parameter space (w,b). Our goal is to find the (w,b) that minimizes $\mathscr{L}$, the lowest point on this error surface.

Figure 1.4 shows a 3D plot of the MSE loss surface as a function of w and b for the example data.

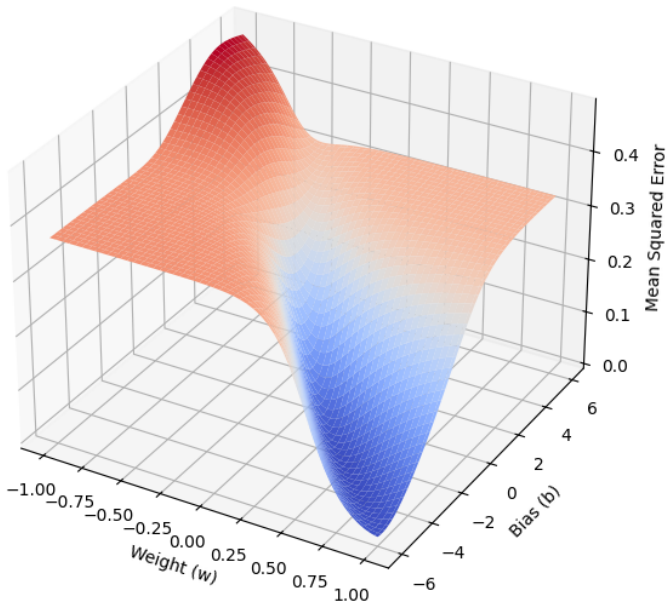


Figure 1.4: 3D Error Surface Over Example Data

1.3.2 Sigmoid Neuron Learning Algorithm: Gradient Descent

Let $\theta = [w, b]$ be the vector of parameters, initialized randomly. Suppose we move in the direction of a change $\Delta\theta = [\Delta w, \Delta b]$. Instead of making a full step, we scale the movement by a small scalar η to stay conservative.

$$\theta_{\text{new}} = \theta + \eta \cdot \Delta\theta$$

What is the right $\Delta\theta$ *to choose?* [1] To answer this, we turn to the Taylor series expansion. Let $u = \Delta\theta$. Then the Taylor expansion of the loss function $\mathscr{L}(\theta)$ around θ is

$$\mathscr{L}(\theta + \eta u) = \mathscr{L}(\theta) + \eta \cdot u^T \nabla_\theta \mathscr{L}(\theta) + \frac{\eta^2}{2!} \cdot u^T \nabla^2 \mathscr{L}(\theta) u + \frac{\eta^3}{3!} \cdot \dots$$

Since η is small, the higher-order terms are negligible.

$$\mathscr{L}(\theta + \eta u) \approx \mathscr{L}(\theta) + \eta \cdot u^T \nabla_\theta \mathscr{L}(\theta)$$

For the move to reduce the loss, we require

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) < 0 \quad \Rightarrow \quad u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

To understand how negative this term can be, consider the angle β between u and $\nabla_\theta \mathscr{L}(\theta)$.

$$\cos(\beta) = \frac{u^T \nabla_\theta \mathscr{L}(\theta)}{\|u\| \cdot \|\nabla_\theta \mathscr{L}(\theta)\|} \text{ and } -1 \leq \cos(\beta) \leq 1$$

Let $k = \|u\| \cdot \|\nabla_\theta \mathscr{L}(\theta)\|$, then

$$-k \leq u^T \nabla_\theta \mathscr{L}(\theta) \leq k$$

This implies that the most negative value is attained when $\cos(\beta) = -1$, i.e., when the angle is $180°$.

$$u = -\nabla_\theta \mathscr{L}(\theta)$$

Hence, the *gradient descent rule* says that the direction u to move in should be opposite to the gradient of the loss.

$$w_{t+1} = w_t - \eta \frac{\partial \mathscr{L}(w,b)}{\partial w}\bigg|_{w=w_t, b=b_t} \quad \text{and} \quad b_{t+1} = b_t - \eta \frac{\partial \mathscr{L}(w,b)}{\partial b}\bigg|_{w=w_t, b=b_t}$$

Gradient Descent Algorithm

$t \leftarrow 0$
max iterations $\leftarrow 1000$
while $t <$ max iterations **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla_w \mathscr{L}(w_t, b_t)$
  $b_{t+1} \leftarrow b_t - \eta \nabla_b \mathscr{L}(w_t, b_t)$
  $t \leftarrow t + 1$
end while

Starting from initial values $w = 0$ and $b = 0$, gradient descent converges to final parameters $w = 0.1487$, $b = -0.4139$ with a final loss of 0.0527 after 100 iterations.

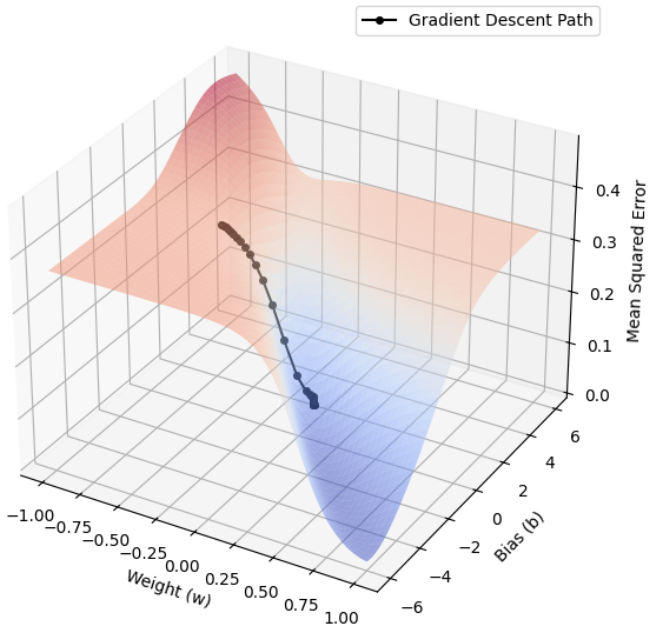


Figure 1.5: Error Surface With Gradient Descent Trajectory.

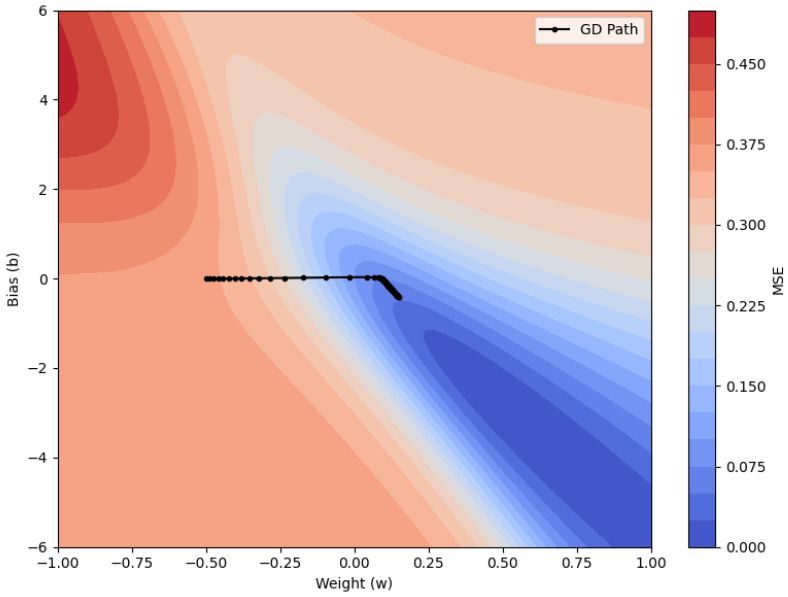


Figure 1.6: Contour Plot of Gradient Descent Trajectory on Error Surface.

Visualizing in three dimensions can get cumbersome. So, we use 2D contour plots instead. When the contours are close together, the slope is steep in that direction.

1.3.3 Momentum Based Gradient Descent

Gradient descent can be slow in regions with gentle slopes. This happens because the gradient becomes very small, leading to small updates. To improve this, we can look at how updates behave over time.

The intuition is that if we keep getting asked to move in the same direction, it's reasonable to trust that direction and take bigger steps — just like a ball gathers speed when rolling downhill.

$$\text{update}_t = \gamma \cdot \text{update}_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - \text{update}_t$$

Here, γ is the momentum coefficient (typically between 0.5 and 0.9), and η is the learning rate. This rule incorporates both the current gradient and the accumulated history of past gradients.

$$\begin{aligned}
\text{update}_0 &= 0 \\
\text{update}_1 &= \gamma \cdot \text{update}_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
\text{update}_2 &= \gamma \cdot \text{update}_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
\text{update}_3 &= \gamma \cdot \text{update}_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
\text{update}_4 &= \gamma \cdot \text{update}_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4
\end{aligned}$$

Continuing this process, we can write the general form,

$$\text{update}_t = \sum_{k=1}^{t} \gamma^{t-k} \cdot \eta \nabla w_k$$

This shows that updates are a weighted sum of all past gradients, with exponentially decaying weights controlled by γ.

Momentum-based Gradient Descent Algorithm

$t \leftarrow 0$
max iterations $\leftarrow 1000$
initialize updates: $u_0^w \leftarrow 0, \quad u_0^b \leftarrow 0$
momentum coefficient $\gamma \in [0, 1)$
learning rate $\eta > 0$
while $t <$ max iterations **do**
  $u_{t+1}^w \leftarrow \gamma \cdot u_t^w + \eta \nabla_w \mathscr{L}(w_t, b_t)$
  $u_{t+1}^b \leftarrow \gamma \cdot u_t^b + \eta \nabla_b \mathscr{L}(w_t, b_t)$
  $w_{t+1} \leftarrow w_t - u_{t+1}^w$
  $b_{t+1} \leftarrow b_t - u_{t+1}^b$
  $t \leftarrow t + 1$
end while

In our example, consider the initial values $w = 0$ and $b = 0$ and $\gamma = 0.9$. Momentum gradient descent converges to final parameters $w = 0.4328$, $b = -2.3303$ with a final loss of 0.004201 after 100 iterations. This surely performs better than the vanilla gradient descent.

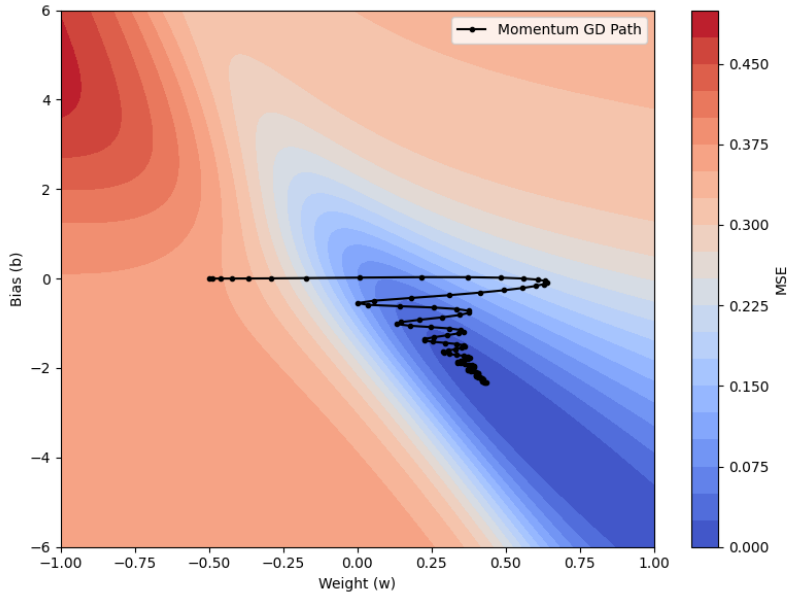


Figure 1.7: Contour Plot of Momentum Gradient Descent Trajectory on Error Surface.

1.3.4 Nesterov Accelerated Gradient Descent

Even in regions with gentle slopes, momentum-based gradient descent takes large steps because momentum carries it forward. It oscillates in and out of the minima valley, often taking many U-turns before finally converging. Despite these oscillations, it converges faster than vanilla gradient descent. But *can we reduce these oscillations?*

The intuition is to *look before we leap*. Recall the momentum update equation,

$$\text{update}_t = \gamma \cdot \text{update}_{t-1} + \eta \nabla_w \mathbf{w}_t$$

We move at least by $\gamma \cdot \text{update}_{t-1}$, plus a bit more by $\eta \nabla_w \mathbf{w}_t$.

What if we calculate the gradient at a look-ahead point instead of the current position? Let's say we define

$$\mathbf{w}_{\text{look ahead}} = \mathbf{w}_t - \gamma \cdot \text{update}_{t-1}$$

Then calculate the gradient $\nabla_w \mathbf{w}_{\text{look ahead}}$ instead of $\nabla_w \mathbf{w}_t$.

Thus, the update rule for Nesterov Accelerated Gradient (NAG) becomes

$$\begin{aligned} \mathbf{w}_{\text{look ahead}} &= \mathbf{w}_t - \gamma \cdot \text{update}_{t-1} \\ \text{update}_t &= \gamma \cdot \text{update}_{t-1} + \eta \nabla_w \mathbf{w}_{\text{look ahead}} \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \text{update}_t \end{aligned}$$

A similar update applies for the bias term b_t.

Nesterov Accelerated Gradient (NAG) Algorithm

$t \leftarrow 0$
max iterations $\leftarrow 1000$
initialize $\text{update}^w_{-1} \leftarrow \mathbf{0}$, $\text{update}^b_{-1} \leftarrow 0$
momentum coefficient $\gamma \in [0,1)$
learning rate $\eta > 0$

while $t <$ max iterations **do**
$\mathbf{w}_{\text{look ahead}} \leftarrow \mathbf{w}_t - \gamma \cdot \text{update}^w_{t-1}$
$b_{\text{look ahead}} \leftarrow b_t - \gamma \cdot \text{update}^b_{t-1}$

$\text{update}^w_t \leftarrow \gamma \cdot \text{update}^w_{t-1} + \eta \nabla_w \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$
$\text{update}^b_t \leftarrow \gamma \cdot \text{update}^b_{t-1} + \eta \nabla_b \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$

$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \text{update}^w_t$
$b_{t+1} \leftarrow b_t - \text{update}^b_t$
$t \leftarrow t+1$
end while

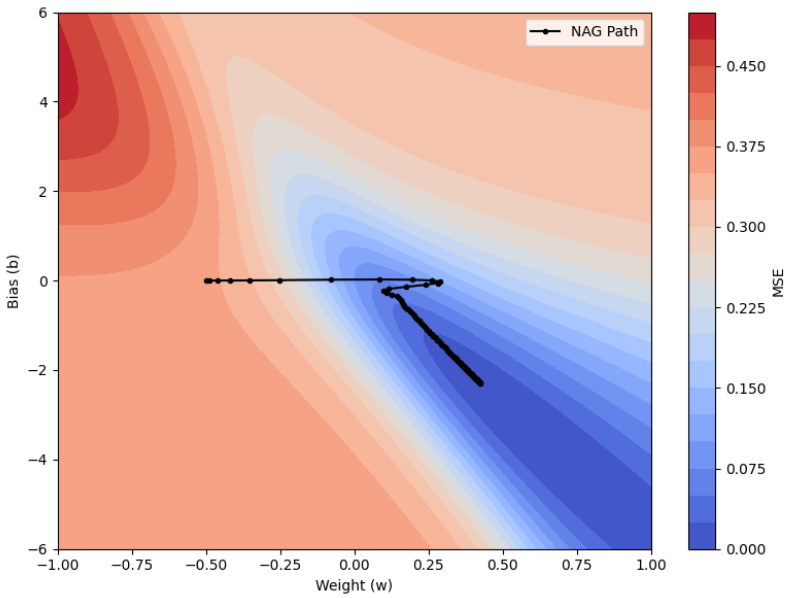


Figure 1.8: Contour Plot of Nesterov Gradient Descent Trajectory on Error Surface.

Looking ahead allows NAG to correct its course faster than momentum-based gradient descent. This reduces oscillations and lowers the chance of escaping the minima valley.

Starting from initial values $w = 0$ and $b = 0$, Nesterov Gradient Descent converges to final parameters $w = 0.4247$, $b = -2.2946$ with a final loss of 0.004477 after 100 iterations.

1.3.5 Stochastic Gradient Descent

Batch gradient descent computes the exact gradient of the loss by averaging the gradients over the entire dataset before updating the parameters. Because the full dataset is used, each update guarantees a decrease in the loss.

However, this method becomes computationally expensive for large datasets. For example, with one million data points, one parameter update requires calculating one million gradients.

Stochastic Gradient Descent (SGD) improves efficiency by updating parameters after evaluating the gradient on each individual data point. Hence, for one million data points, SGD performs one million updates per epoch, where an *epoch* is defined as a full pass over the dataset.

However, the gradient used in SGD is a noisy, stochastic estimate of the true gradient since it is computed using only one data point rather than the entire dataset. Due to this noise, there is no guarantee that each SGD update reduces the overall loss function. Indeed, the updates tend to oscillate, especially in small datasets, as each data point tries to optimize locally without considering the global effect on other points.

Even with these jumps, SGD can still find a good solution in the long run, as long as the learning rate is adjusted properly over time.

To smooth things out, we can use mini-batches. Instead of just one point, we use a small group of data points to compute the gradient. This reduces the noise while keeping the updates fast. It also helps the model converge more smoothly.

Also, the randomness in SGD isn't always bad. It can help the model escape flat spots or shallow traps in the loss surface, which sometimes leads to better results.

Mini-Batch Stochastic Gradient Descent Algorithm

$t \leftarrow 0$
max iterations $\leftarrow 1000$
mini-batch size $\leftarrow m$

while $t <$ max iterations **do**
  Sample mini-batch $\mathscr{B}_t$ of size m from training data

  Compute gradients over mini-batch:
    $g_w \leftarrow \frac{1}{m} \sum_{(x_i,y_i)\in\mathscr{B}_t} \nabla_w \mathscr{L}(w_t, b_t;\ x_i, y_i)$
    $g_b \leftarrow \frac{1}{m} \sum_{(x_i,y_i)\in\mathscr{B}_t} \nabla_b \mathscr{L}(w_t, b_t;\ x_i, y_i)$

    $w_{t+1} \leftarrow w_t - \eta g_w$
    $b_{t+1} \leftarrow b_t - \eta g_b$

  $t \leftarrow t+1$
end while

Starting from initial values $w = 0$ and $b = 0$, Stochastic Gradient Descent converges to final parameters $w = 0.3706$, $b = -1.9689$ with a final loss of 0.007861 after 100 iterations.

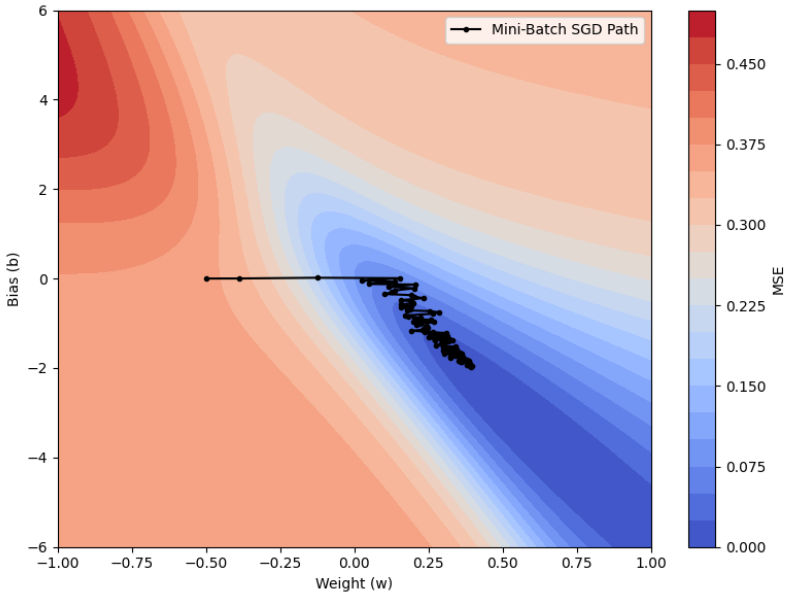


Figure 1.9: Contour Plot of Stochastic Gradient Descent Trajectory on Error Surface.

1.3.6 Gradient Descent with Adaptive Learning

One might think to solve the problem of navigating gentle slopes by setting a high learning rate η, effectively amplifying small gradients. But it is better to have a learning rate that adapts to the gradient itself.

Intuition 1 (Adagrad): Decay the learning rate for each parameter based on its past updates. More updates lead to more decay. The update rule for Adagrad is

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \cdot \nabla w_t$$

A similar set of equations applies for the bias parameter b_t.

Batch Gradient Descent with Adagrad

$t \leftarrow 0$, max iterations $\leftarrow 1000$
$v_w \leftarrow 0$, $v_b \leftarrow 0$
while $t <$ max iterations **do**
  $g_{w_t} \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$, $g_{b_t} \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$
  $v_w \leftarrow v_w + g_{w_t}^2$, $v_b \leftarrow v_b + g_{b_t}^2$
  $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{v_w}+\varepsilon} \cdot g_{w_t}$, $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{v_b}+\varepsilon} \cdot g_{b_t}$
  $t \leftarrow t + 1$
end while

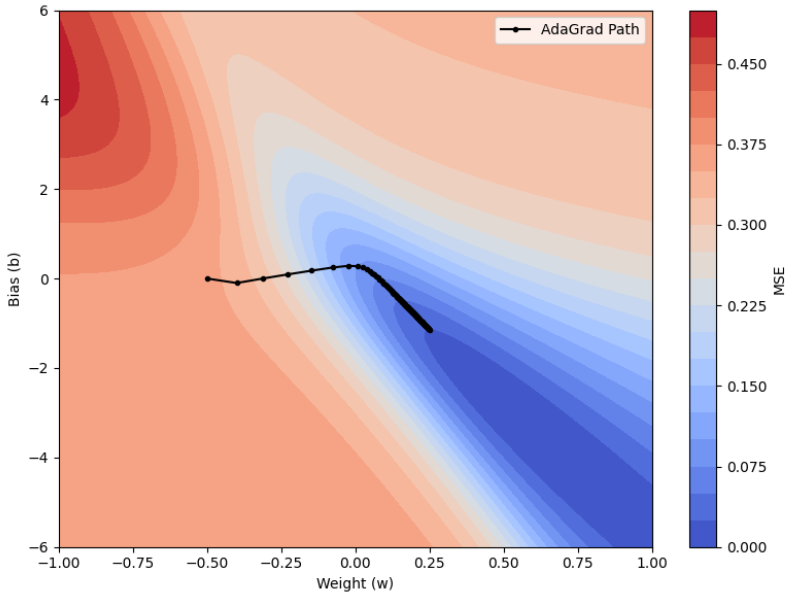


Figure 1.10: Contour Plot of Gradient Descent Trajectory with Adagrad Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with Adagrad Learning converges to final parameters $w = 0.2493$, $b = -1.1415$ with a final loss of 0.024577 after 100 iterations.

Intuition 2 (RMSProp): Adagrad decays the learning rate too aggressively. After some time, frequently updated parameters get very small updates because the denominator $\sqrt{v_t}$ grows without bound. To fix this, RMSProp decays v_t to prevent its rapid growth.

The update rule for RMSProp is

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \cdot \nabla w_t$$

Again, a similar set of equations applies for b_t.

Batch Gradient Descent with RMSProp

$t \leftarrow 0$, max iterations $\leftarrow 1000$
Initialize $v_w \leftarrow 0$, $v_b \leftarrow 0$
Choose η, β, ε
while $t <$ max iterations **do**
  $g_w \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$, $g_b \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$
  $v_w \leftarrow \beta \cdot v_w + (1-\beta) \cdot g_w^2$, $v_b \leftarrow \beta \cdot v_b + (1-\beta) \cdot g_b^2$
  $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{v_w}+\varepsilon} \cdot g_w$, $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{v_b}+\varepsilon} \cdot g_b$
  $t \leftarrow t+1$
end while

Adagrad gets stuck near convergence because its learning rate decays too much, making it hard to move in some directions (like the vertical b direction). RMSProp fixes this by decaying v_t less aggressively.

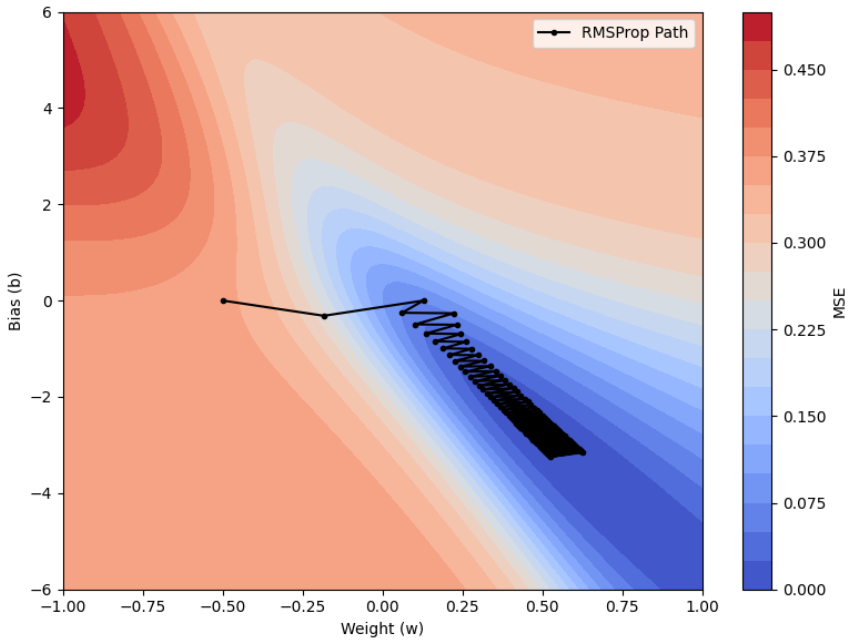


Figure 1.11: Contour Plot of Gradient Descent Trajectory with RMSProp Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with RMSProp Learning converges to final parameters $w = 0.6249$, $b = -3.1551$ with a final loss of 0.005095 after 100 iterations.

Intuition 3 (Adam): Adam builds on RMSProp by also keeping a cumulative history of gradients (first moment estimate).

The update rules for Adam are

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla w_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \cdot \hat{m}_t$$

A similar set of equations applies for b_t.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with Adam Learning converges to final parameters $w = 0.5227$, $b = -2.9403$ with a final loss of 0.001289 after 100 iterations.

Batch Gradient Descent with Adam

$t \leftarrow 0$, max iterations $\leftarrow 1000$
Initialize $m_w = 0,\ v_w = 0,\ m_b = 0,\ v_b = 0$
Choose β_1, β_2

while $t <$ max iterations **do**
Compute gradients: $g_w \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$, $g_b \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$

$m_w \leftarrow \beta_1 \cdot m_w + (1-\beta_1) \cdot g_w$, $v_w \leftarrow \beta_2 \cdot v_w + (1-\beta_2) \cdot g_w^2$
$\hat{m}_w \leftarrow \frac{m_w}{1-\beta_1^t}$, $\hat{v}_w \leftarrow \frac{v_w}{1-\beta_2^t}$
$w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{\hat{v}_w}+\varepsilon} \cdot \hat{m}_w$

$m_b \leftarrow \beta_1 \cdot m_b + (1-\beta_1) \cdot g_b$, $v_b \leftarrow \beta_2 \cdot v_b + (1-\beta_2) \cdot g_b^2$
$\hat{m}_b \leftarrow \frac{m_b}{1-\beta_1^t}$, $\hat{v}_b \leftarrow \frac{v_b}{1-\beta_2^t}$
$b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{\hat{v}_b}+\varepsilon} \cdot \hat{m}_b$

$t \leftarrow t+1$
end while

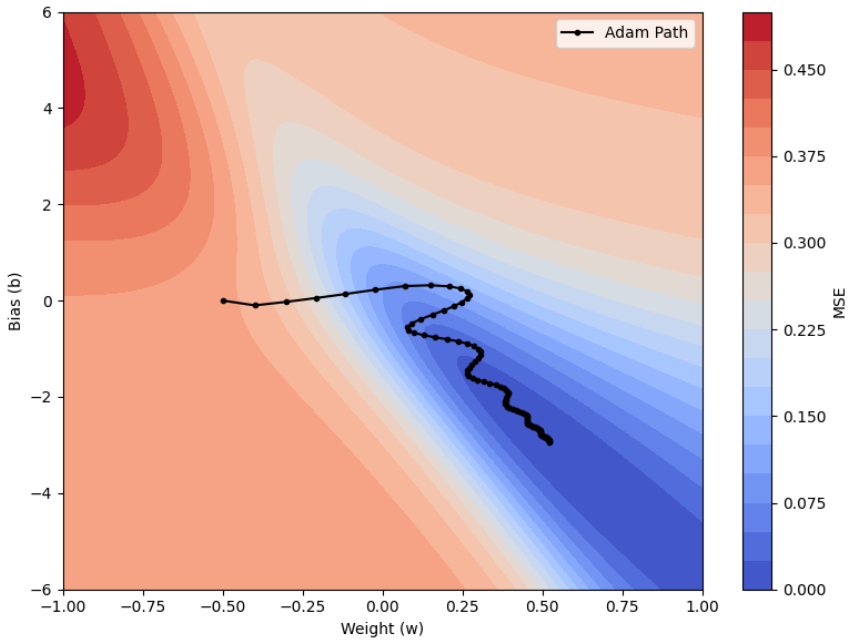


Figure 1.12: Contour Plot of Gradient Descent Trajectory with Adam Learning on Error Surface.

All these adaptive learning rate methods modify the vanilla gradient descent update. Adam can also be combined with Nesterov's lookahead method for further improvements.

Nesterov-Adam (Nadam) Algorithm

$t \leftarrow 0$, max iterations $\leftarrow 1000$
Initialize $m_w = 0,\ v_w = 0,\ m_b = 0,\ v_b = 0$
Initialize $w_0,\ b_0$, momentum coefficient $\gamma = \beta_1 \in [0,1)$, learning rate $\eta > 0$
while $t <$ max iterations **do**

$\mathbf{w}_{\text{look ahead}} \leftarrow \mathbf{w}_t - \gamma \cdot m_w$
$b_{\text{look ahead}} \leftarrow b_t - \gamma \cdot m_b$
Compute gradients: $g_w \leftarrow \nabla_w \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$, $g_b \leftarrow \nabla_b \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$

$m_w \leftarrow \beta_1 \cdot m_w + (1-\beta_1) \cdot g_w$, $v_w \leftarrow \beta_2 \cdot v_w + (1-\beta_2) \cdot g_w^2$
$\hat{m}_w \leftarrow \frac{m_w}{1-\beta_1^{t+1}}$, $\hat{v}_w \leftarrow \frac{v_w}{1-\beta_2^{t+1}}$
$w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{\hat{v}_w}+\varepsilon} \cdot (\gamma \cdot \hat{m}_w + (1-\gamma) \cdot g_w)$

$m_b \leftarrow \beta_1 \cdot m_b + (1-\beta_1) \cdot g_b$, $v_b \leftarrow \beta_2 \cdot v_b + (1-\beta_2) \cdot g_b^2$
$\hat{m}_b \leftarrow \frac{m_b}{1-\beta_1^{t+1}}$, $\hat{v}_b \leftarrow \frac{v_b}{1-\beta_2^{t+1}}$
$b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{\hat{v}_b}+\varepsilon} \cdot (\gamma \cdot \hat{m}_b + (1-\gamma) \cdot g_b)$
$t \leftarrow t+1$
end while

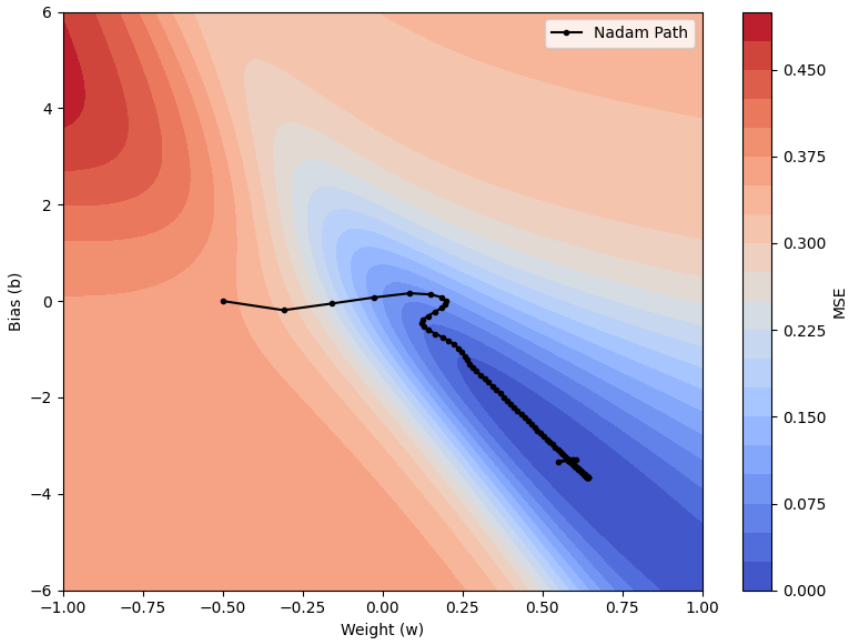


Figure 1.13: Contour Plot of Nesterov Gradient Descent Trajectory with Adam Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Nesterov Gradient Descent with Adam Learning converges to final parameters $w = 0.5829$, $b = -3.3043$ with a final loss of 0.000833 after 100 iterations.

Method	Final Parameters (w, b)	Final Loss
Vanilla Gradient Descent	$w = 0.1487,\ b = -0.4139$	0.0527
Momentum Gradient Descent ($\gamma = 0.9$)	$w = 0.4328,\ b = -2.3303$	0.004201
Nesterov Gradient Descent	$w = 0.4247,\ b = -2.2946$	0.004477
Stochastic Gradient Descent	$w = 0.3706,\ b = -1.9689$	0.007861
Adagrad Gradient Descent	$w = 0.2493,\ b = -1.1415$	0.024577
RMSProp Gradient Descent	$w = 0.6249,\ b = -3.1551$	0.005095
Adam Gradient Descent	$w = 0.5227,\ b = -2.9403$	0.001289
Nesterov with Adam (Best)	$w = 0.5829,\ b = -3.3043$	**0.000833**

Table 1.4: Comparison of Gradient Descent Methods (Initial values: $w = 0$, $b = 0$, 100 iterations)

1.3.7 Layer of Sigmoid Neurons

Recall a multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors). Similarly, the representation power of a multilayer network of sigmoid neurons is given by the **Universal Approximation Theorem.** To prove this, we will need the result from **Stone-Weierstrass Theorem**.

Theorem 1.3. ***Stone-Weierstrass Theorem****. Let K be a compact subset of $\mathbb{R}^n$ and let $\mathscr{A}$ be a subalgebra of $C(K)$ (the space of continuous real-valued functions on K) such that*

1. *$\mathscr{A}$ separates points of K (i.e., for any $x, y \in K$ with $x \neq y$, there exists $f \in \mathscr{A}$ such that $f(x) \neq f(y)$)*
2. *$\mathscr{A}$ contains the constant functions*

Then $\mathscr{A}$ is dense in $C(K)$ with respect to the uniform norm.

Imagine you have a collection of continuous functions defined on a *nice* and *closed* space (like a closed and bounded interval). If this collection of functions has two key properties:

1. *They can tell points apart*. For any two distinct points in your space, you can find a function in your collection that gives different values for those two points.
2. *They include all the constant functions*. You have functions in your collection that are just a fixed number everywhere.

Then, you can use combinations of these functions (adding them, multiplying them, multiplying them by constants) to get arbitrarily close to any other continuous function on that space.

Essentially, if your initial collection of functions is rich enough to distinguish between points and includes constants, it's *dense* enough to build approximations for any other continuous function you can think of on that space.

Proof. Let $f \in C(K)$ and $\varepsilon > 0$. We aim to show that there exists a function $g \in \mathscr{A}$ such that

$$\|f - g\|_\infty < \varepsilon.$$

Step 1: Approximate indicator functions.

Fix $x_0 \in K$. For any $x \in K$, since $\mathscr{A}$ separates points, there exists $\phi_x \in \mathscr{A}$ such that $\phi_x(x) \neq \phi_x(x_0)$. Define:

$$\psi_x(y) = \left(\frac{\phi_x(y) - \phi_x(x_0)}{\phi_x(x) - \phi_x(x_0)} \right)^2.$$

Then $\psi_x \in \mathscr{A}$ because $\mathscr{A}$ is a subalgebra (closed under addition, multiplication, and scalar multiplication), and we've used only these operations. Also, $\psi_x(x_0) = 0$, and $\psi_x(x) = 1$.

Now, for a finite set $\{x_1, \ldots, x_m\} \subset K$, we can construct a partition of unity-like approximation by defining functions $\{\psi_{x_j}\}_{j=1}^m$ and then normalize

$$S(y) = \sum_{j=1}^m \psi_{x_j}(y), \quad \rho_j(y) = \frac{\psi_{x_j}(y)}{S(y)}.$$

Each $\rho_j \in \mathscr{A}$, and $\sum_{j=1}^m \rho_j(y) = 1$ for all $y \in K$.

Step 2: Local approximation of f.

Since f is uniformly continuous on compact K, there exists a finite δ-net $\{x_1, \ldots, x_m\} \subset K$ such that for all $y \in K$, there exists x_j with $|f(y) - f(x_j)| < \varepsilon$.

Now define the function

$$g(y) = \sum_{j=1}^m f(x_j)\rho_j(y).$$

Then $g \in \mathscr{A}$ because $\rho_j \in \mathscr{A}$ and $f(x_j)$ are constants (constants are in $\mathscr{A}$, and the algebra is closed under scalar multiplication and addition).

Step 3: Approximation bound.

We estimate the uniform difference

$$|f(y) - g(y)| = \left| f(y) - \sum_{j=1}^m f(x_j)\rho_j(y) \right| = \left| \sum_{j=1}^m \rho_j(y)(f(y) - f(x_j)) \right|.$$

Using the triangle inequality

$$|f(y) - g(y)| \leq \sum_{j=1}^m \rho_j(y)|f(y) - f(x_j)|.$$

Since each $\rho_j(y) \geq 0$, and $\sum \rho_j(y) = 1$, this is a convex combination of errors $|f(y) - f(x_j)|$, each less than ε. So,

$$|f(y) - g(y)| < \varepsilon \quad \text{for all } y \in K.$$

Hence,

$$\|f - g\|_\infty < \varepsilon.$$

□

Example 1.1. *Polynomials can be used to approximate any continuous function on a closed interval.*

Solution 1.1. Let K be a compact subset of $\mathbb{R}^n$. We consider the subalgebra $\mathscr{A}$ of $C(K)$ consisting of all polynomial functions on K. For clarity, let's first consider the common case where K is a closed and bounded interval in $\mathbb{R}$, say $K = [a,b]$.

A function $f \in \mathscr{A}$ has the form $f(x) = c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0$ for some non-negative integer m and real coefficients $c_0, c_1, \ldots, c_m$.

We need to demonstrate that $\mathscr{A}$ satisfies the two properties required by the Stone-Weierstrass Theorem.

1. $\mathscr{A}$ separates points of K

This property requires that for any distinct points $x, y \in K$ (i.e., $x \neq y$), there must exist a function $f \in \mathscr{A}$ such that $f(x) \neq f(y)$.

Consider any two distinct points $x, y \in [a,b]$ such that $x \neq y$. Let's choose the polynomial function $f(t) = t$. This is a simple polynomial of degree 1 (where $m = 1$, $c_1 = 1$, and $c_0 = 0$). Evaluating this function at x and y, we get $f(x) = x$ and $f(y) = y$. Since we initially assumed $x \neq y$, it directly follows that $f(x) \neq f(y)$.

Therefore, the set of polynomial functions $\mathscr{A}$ successfully separates points of K.

2. $\mathscr{A}$ contains the constant functions

This property requires that for any real number c, the constant function $g(t) = c$ for all $t \in K$ must be an element of $\mathscr{A}$.

A constant function $g(t) = c$ can be expressed as a polynomial of degree 0. We can write $g(t) = c \cdot t^0$, or simply $g(t) = c$. This fits the general form of a polynomial where $m = 0$ and $c_0 = c$.

Thus, for any real number c, the constant function $g(t) = c$ is indeed a polynomial.

Therefore, the set of polynomial functions $\mathscr{A}$ contains all constant functions.

Since the set of polynomial functions on $K = [a,b]$ satisfies both of these crucial properties, the Stone-Weierstrass Theorem implies that the set of polynomials is **dense** in $C([a,b])$.

This is a fundamental result, meaning that any continuous real-valued function defined on a closed and bounded interval can be uniformly approximated arbitrarily well by polynomials. The argument extends naturally to compact subsets of $\mathbb{R}^n$. In this case, $\mathscr{A}$ would comprise polynomials in n variables.

Refer figure 1.14 for illustration.

Theorem 1.4. ***Universal Approximation Theorem.*** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded, and continuous function (such as the sigmoid function). Let $K \subset \mathbb{R}^n$ be compact. Then for any continuous function $f : K \to \mathbb{R}$ and any $\varepsilon > 0$, there exist $N \in \mathbb{N}$, weights $\alpha_1, \ldots, \alpha_N \in \mathbb{R}$, weight vectors $\mathbf{w}_1, \ldots, \mathbf{w}_N \in \mathbb{R}^n$, and biases $b_1, \ldots, b_N \in \mathbb{R}$ such that the function*

$$g(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

satisfies $\sup_{\mathbf{x} \in K} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon$.

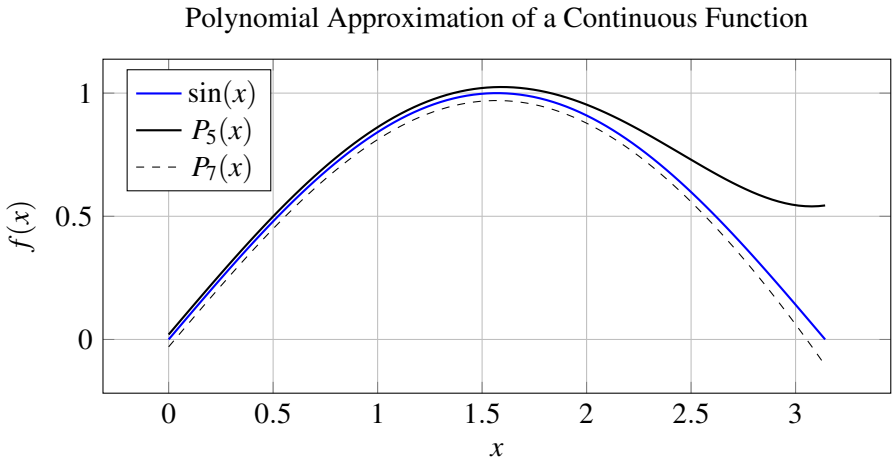


Figure 1.14: Illustration of the Stone-Weierstrass theorem. Polynomials $P_5(x)$ and $P_7(x)$ successively approximate $\sin(x)$ with increasing accuracy on the compact interval $[0, \pi]$.

Proof. We want to show that the set of functions

$$\mathscr{F} = \left\{ \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j) : N \in \mathbb{N},\ \alpha_j \in \mathbb{R},\ \mathbf{w}_j \in \mathbb{R}^n,\ b_j \in \mathbb{R} \right\}$$

is dense in $C(K)$, the space of continuous real-valued functions on compact $K \subset \mathbb{R}^n$, with respect to the uniform norm. Let us define

$$\mathscr{A} = \text{span}\{\sigma(\mathbf{w}^T \mathbf{x} + b) : \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}\},$$

which is a subalgebra of $C(K)$.

Step 1: Closure under addition, multiplication, and scalar multiplication.

The set $\mathscr{A}$ is a linear span of compositions of affine functions with σ, so it is closed under addition and scalar multiplication by construction.

Step 2: Separation of points.

Let $\mathbf{x}, \mathbf{y} \in K$ with $\mathbf{x} \neq \mathbf{y}$. Since σ is non-constant and continuous, there exists a hyperplane (i.e., choice of $\mathbf{w} \in \mathbb{R}^n$) such that $\mathbf{w}^T \mathbf{x} \neq \mathbf{w}^T \mathbf{y}$. Then for any fixed b, we have

$$\sigma(\mathbf{w}^T \mathbf{x} + b) \neq \sigma(\mathbf{w}^T \mathbf{y} + b),$$

because σ is strictly increasing (as in sigmoid), or at least non-constant and continuous, so it distinguishes between different inputs. Hence, $\mathscr{A}$ separates points of K.

Step 3: Constants are in $\mathscr{A}$.

Since σ is bounded and non-constant, it attains at least two values. For large positive or negative arguments, $\sigma(z) \to c_1$ or c_2, so we can scale and shift σ to approximate constant functions arbitrarily well. More directly, linear combinations of shifted sigmoids can approximate any constant function. Hence, $\mathscr{A}$ contains constant functions or can approximate them arbitrarily well, which is enough for density in the uniform norm.

By the Stone-Weierstrass Theorem 1.3, $\mathscr{A}$ is dense in $C(K)$. Thus, for any $f \in C(K)$ and $\varepsilon > 0$, there exists a finite sum of the form

$$g(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

such that

$$\sup_{\mathbf{x} \in K} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon.$$

□

Corollary 1.1. *A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.*

In other words, there is a guarantee that for any function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, we can always find a neural network (with one hidden layer containing enough sigmoid neurons) whose output $g(\mathbf{x})$ satisfies $|g(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$.

1.4 Neural Network

In earlier sections, we saw how a single layer of perceptrons or sigmoid neurons can represent any Boolean function or real function respectively. But there's a catch: the number of neurons needed grows exponentially with the number of inputs. That makes single-layer models impractical for real-world tasks.

Things changed in the 1980s. ***Geoffrey Hinton***, ***David Rumelhart***, and ***Ronald Williams*** proposed a way to train networks with multiple hidden layers. Their work introduced the backpropagation algorithm [3], which made deep learning possible again. It allowed neural networks to be trained efficiently, layer by layer, even when stacked deep. This breakthrough helped revive interest in neural networks after a long period of disinterest in the field.

1.4.1 Feedforward Neural Network Architecture

We start with an input vector $\mathbf{x} \in \mathbb{R}^n$, where n is the number of input features. This input passes through a sequence of layers in the network. A typical feedforward neural network has L layers in total, $L-1$ hidden layers and one output layer.

Each hidden layer contains a certain number of neurons (often close to n), and every layer performs a transformation on the input it receives from the previous layer. For instance, $L = 3$ in figure 1.15, as we have two hidden layers followed by an output layer.

Structure of a Neuron

Each neuron in the hidden and output layers operates in two steps:

- ***Pre-activation:*** This is a linear transformation of the input from the previous layer. It combines the inputs using weights and adds a bias term. For the i^{th} layer, we denote this intermediate result by $\mathbf{a}^{(i)} \in \mathbb{R}^n$.
- ***Activation:*** This is a non-linear transformation of the pre-activation. The output is denoted by $\mathbf{h}^{(i)} \in \mathbb{R}^n$.

This split into pre-activation and activation helps separate the roles of the linear and non-linear parts of the computation. The linear step captures how inputs are combined, while the non-linear step introduces flexibility. Without non-linearity, the entire network would collapse into a single linear transformation, no matter how many layers we stack.

Functions like sigmoid, tanh, and ReLU are used as activation functions. These functions are chosen because they are simple, differentiable, and behave differently for different inputs. This helps the network respond in varied ways, which is essential for learning complex patterns from data.

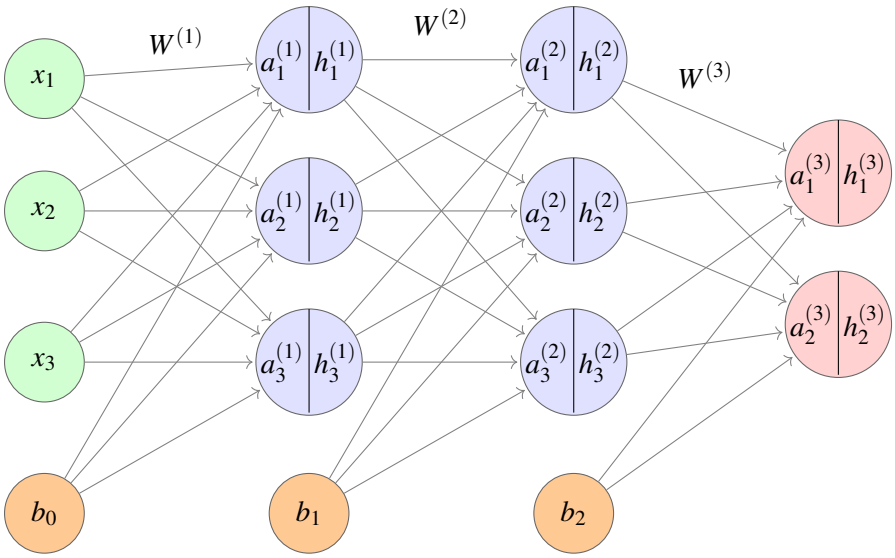


Figure 1.15: Feedforward Neural Network Architecture

Layer Computation

Let $\mathbf{h}^{(0)} = \mathbf{x}$, the input vector. For any hidden layer $i \in \{1, \ldots, L-1\}$, the operations in the layer are

$$\mathbf{a}^{(i)} = \mathbf{b}^{(i)} + \mathbf{W}^{(i)}\mathbf{h}^{(i-1)} \quad \text{and} \quad \mathbf{h}^{(i)} = g(\mathbf{a}^{(i)}).$$

Here,

- $\mathbf{W}^{(i)} \in \mathbb{R}^{n \times n}$ is the weight matrix for layer i,
- $\mathbf{b}^{(i)} \in \mathbb{R}^{n}$ is the bias vector for layer i,
- g is the activation function applied to each component of $\mathbf{a}^{(i)}$.

At the final (output) layer, labeled as layer L, the activation is:

$$\mathbf{a}^{(L)} = \mathbf{b}^{(L)} + \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} \quad \text{and} \quad f(\mathbf{x}) = \mathbf{h}^{(L)} = O(\mathbf{a}^{(L)}),$$

where O is an output-specific activation function. For example, softmax is used in classification tasks to assign probabilities to each class, while identity or linear functions are used in regression tasks to produce continuous outputs.

1.4.2 A Typical Supervised Machine Learning Setup

We now place the feedforward architecture in the context of supervised learning, where the goal is to learn from labeled data to make predictions.

- **Data:** We are given a dataset of N examples
$$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$$
where $\mathbf{x}_i \in \mathbb{R}^n$ is the input vector and $\mathbf{y}_i \in \mathbb{R}^k$ is the target output.
- **Model:** We use the feedforward neural network to define a function $f(\cdot)$ that maps inputs to outputs. The predicted output for $\mathbf{x}_i$ is
$$\hat{\mathbf{y}}_i = f(\mathbf{x}_i) = O\left(\mathbf{W}^{(3)} g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}\right)$$
where
  - $\mathbf{W}^{(\ell)}$, $\mathbf{b}^{(\ell)}$ are weights and biases for layer ℓ,
  - g is the non-linear activation in hidden layers,
  - O is the output activation function.
- **Parameters:** The learnable parameters of the network are the weights and biases.
$$\theta = \left\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \mathbf{b}^{(3)}\right\}$$
- **Loss Function:** To measure how well the model is performing, we define an objective function. A common choice is the Mean Squared Error (MSE).
$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} (\hat{y}_{ij} - y_{ij})^2$$
More generally, this can be written as
$$\min_{\theta} \mathscr{L}(\theta)$$
where $\mathscr{L}(\theta)$ is any suitable loss function that depends on the model's parameters.
- **Learning Algorithm:** The parameters θ are updated to minimize the loss using gradient-based optimization. Gradients are computed using a procedure called backpropagation, and a popular choice for optimization is gradient descent.

The choice of activation and loss functions depends on the nature of the output. For regression tasks with real-valued outputs, a linear activation and squared error loss are common. For classification tasks, where the goal is to assign probabilities to classes, softmax activation combined with cross-entropy loss is a standard choice.

Outputs	Output Activation	Loss Function
Real Values	Linear	Squared Error
Probabilities	Softmax	Cross Entropy

1.4.3 Neural Network Learning Algorithm: Backpropagation

To understand how learning happens in a feedforward neural network, we study how the loss function's error signal flows backward from the output to the input. This is the essence of *backpropagation*. Instead of considering the full network at once, let us pick a specific path first.

We will trace how the error at the output neuron $h_1^{(3)}$ flows backward and contributes to the update of the weight $w_{11}^{(1)}$, the connection from input x_1 to the first neuron in the first hidden layer.

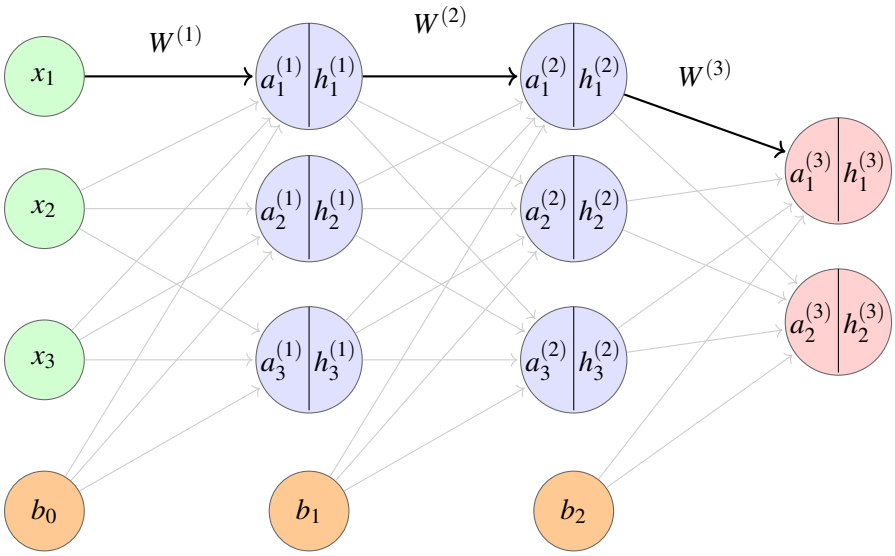


Figure 1.16: Backpropagation of Error Signal Along One Path: $x_1 \to h_1^{(1)} \to h_1^{(2)} \to h_1^{(3)}$

We want to compute how the loss $\mathscr{L}$ changes with respect to the weight $w_{11}^{(1)}$, which connects x_1 to the first neuron in the first hidden layer.

Using the chain rule,

$$\frac{\partial \mathscr{L}}{\partial w_{11}^{(1)}} = \underbrace{\frac{\partial \mathscr{L}}{\partial h_1^{(3)}}}_{\substack{\text{Change of loss} \\ \text{w.r.t. output}}} \cdot \underbrace{\frac{\partial h_1^{(3)}}{\partial a_1^{(3)}}}_{\substack{\text{Change of output} \\ \text{w.r.t. pre-activation}}} \cdot \underbrace{\frac{\partial a_1^{(3)}}{\partial h_1^{(2)}}}_{\substack{\text{Weight} \\ w_{11}^{(3)}}} \cdot \underbrace{\frac{\partial h_1^{(2)}}{\partial a_1^{(2)}}}_{\substack{\text{Change of hidden output} \\ \text{w.r.t. pre-activation}}} \cdot \underbrace{\frac{\partial a_1^{(2)}}{\partial h_1^{(1)}}}_{\substack{\text{Weight} \\ w_{11}^{(2)}}} \cdot \underbrace{\frac{\partial h_1^{(1)}}{\partial a_1^{(1)}}}_{\substack{\text{Change of hidden output} \\ \text{w.r.t. pre-activation}}} \cdot \underbrace{\frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}}_{\substack{\text{Change of pre-activation} \\ \text{w.r.t. weight = input } x_1}}$$

$$\frac{\partial \mathscr{L}}{\partial w_{11}^{(1)}} = \underbrace{\left(\frac{\partial \mathscr{L}}{\partial h_1^{(3)}} \cdot O'(a_1^{(3)}) \right)}_{\delta_1^{(3)} \text{ (error signal at output)}} \cdot w_{11}^{(3)} \cdot g'(a_1^{(2)}) \cdot w_{11}^{(2)} \cdot g'(a_1^{(1)}) \cdot x_1$$

This expression shows how the error signal at the output is scaled and transmitted backward through each layer, weighted by the derivatives of the activations and the weights on the forward path.

To compute the gradients for the entire network, we follow a layer-wise application of the chain rule, moving backward from the output layer to the input layer. The idea is to reuse the intermediate gradient computation, specifically the error signals at each layer, denoted as $\delta^{(l)}$, to efficiently update all the learnable parameters.

Step 1: Gradients w.r.t. Output Layer

We begin at the output layer L, where the error signal is given by

$$\delta^{(L)} = \nabla_{\mathbf{h}^{(L)}} \mathscr{L} \odot g'(\mathbf{a}^{(L)})$$

This captures how the loss changes with respect to the pre-activation at the output, scaled by the derivative of the output activation function. Let's expand this expression to understand the computation element-wise.

Suppose the output layer L has m neurons. Then the output vector is

$$\mathbf{h}^{(L)} = \begin{bmatrix} h_1^{(L)} \\ h_2^{(L)} \\ \vdots \\ h_m^{(L)} \end{bmatrix}, \quad \text{with pre-activations} \quad \mathbf{a}^{(L)} = \begin{bmatrix} a_1^{(L)} \\ a_2^{(L)} \\ \vdots \\ a_m^{(L)} \end{bmatrix}$$

The derivative of the loss with respect to the output vector is

$$\nabla_{\mathbf{h}^{(L)}} \mathscr{L} = \underbrace{\begin{bmatrix} \frac{\partial \mathscr{L}}{\partial h_1^{(L)}} \\ \frac{\partial \mathscr{L}}{\partial h_2^{(L)}} \\ \vdots \\ \frac{\partial \mathscr{L}}{\partial h_m^{(L)}} \end{bmatrix}}_{\text{Sensitivity of loss to network output}} \in \mathbb{R}^m$$

The derivative of the activation function applied element-wise is

$$g'(\mathbf{a}^{(L)}) = \underbrace{\begin{bmatrix} g'(a_1^{(L)}) \\ g'(a_2^{(L)}) \\ \vdots \\ g'(a_m^{(L)}) \end{bmatrix}}_{\text{Local slope of activation function}} \in \mathbb{R}^m$$

Taking the Hadamard product (element-wise multiplication), we compute

$$\delta^{(L)} = \begin{bmatrix} \frac{\partial \mathscr{L}}{\partial h_1^{(L)}} \cdot g'(a_1^{(L)}) \\ \frac{\partial \mathscr{L}}{\partial h_2^{(L)}} \cdot g'(a_2^{(L)}) \\ \vdots \\ \frac{\partial \mathscr{L}}{\partial h_m^{(L)}} \cdot g'(a_m^{(L)}) \end{bmatrix} = \begin{bmatrix} \delta_1^{(L)} \\ \delta_2^{(L)} \\ \vdots \\ \delta_m^{(L)} \end{bmatrix}$$

Each element in this error signal vector is

$$\delta_i^{(L)} = \underbrace{\frac{\partial \mathscr{L}}{\partial h_i^{(L)}}}_{\substack{\text{Loss sensitivity} \\ \text{to output}}} \cdot \underbrace{g'(a_i^{(L)})}_{\substack{\text{Slope of} \\ \text{activation function}}} \qquad \text{for } i = 1, 2, \ldots, m$$

Hence, the vector $\delta^{(L)} \in \mathbb{R}^m$ captures the element-wise product of the gradient of the loss with respect to the network's output and the gradient of the output activation function.

Step 2: Gradients w.r.t. Hidden Layers

For hidden layers $l = L-1, L-2, \ldots, 1$, the error signal is propagated backwards using the recursive formula

$$\delta^{(l)} = \left(\mathbf{W}^{(l+1)^\top} \delta^{(l+1)} \right) \odot g'(\mathbf{a}^{(l)})$$

This formulation ensures efficient reuse of the already-computed error signal from the layer ahead, weighted by the transpose of the weight matrix $\mathbf{W}^{(l+1)}$, and modulated by the local derivative of the activation function at layer l.

Again, let's write this component-wise. The j-th component of $\delta^{(l)}$, for $j = 1, 2, \ldots, n_l$, is given by

$$\delta_j^{(l)} = \underbrace{g'\left(a_j^{(l)}\right)}_{\text{sensitivity of neuron } j \text{ in layer } l} \cdot \underbrace{\sum_{k=1}^{n_{l+1}} w_{kj}^{(l+1)} \cdot \delta_k^{(l+1)}}_{\text{weighted sum of error signals from next layer}}$$

That is, each component of $\delta^{(l)}$ is computed as

$$\delta_j^{(l)} = g'\left(a_j^{(l)}\right) \cdot \left(w_{1j}^{(l+1)} \cdot \delta_1^{(l+1)} + w_{2j}^{(l+1)} \cdot \delta_2^{(l+1)} + \cdots + w_{n_{l+1},j}^{(l+1)} \cdot \delta_{n_{l+1}}^{(l+1)} \right)$$

Expanding the entire vector form gives

$$\delta^{(l)} = \underbrace{\begin{bmatrix} g'\left(a_1^{(l)}\right) \\ g'\left(a_2^{(l)}\right) \\ \vdots \\ g'\left(a_{n_l}^{(l)}\right) \end{bmatrix}}_{\substack{\text{Elementwise derivative of} \\ \text{activations at layer } l}} \odot \underbrace{\begin{bmatrix} \sum_{k=1}^{n_{l+1}} w_{k1}^{(l+1)} \delta_k^{(l+1)} \\ \sum_{k=1}^{n_{l+1}} w_{k2}^{(l+1)} \delta_k^{(l+1)} \\ \vdots \\ \sum_{k=1}^{n_{l+1}} w_{k,n_l}^{(l+1)} \delta_k^{(l+1)} \end{bmatrix}}_{\substack{\left(\mathbf{W}^{(l+1)^\top} \delta^{(l+1)}\right) \\ \text{in } \mathbb{R}^{n_l}}}$$

Step 3: Gradients w.r.t. Parameters

Once we have computed $\delta^{(l)}$ for each layer, the gradients of the loss with respect to the weights and biases follow directly.

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \cdot \mathbf{h}^{(l-1)^\top} \quad , \quad \frac{\partial \mathscr{L}}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$$

where $\mathbf{h}^{(l-1)}$ is the output from the previous layer (or the input vector $\mathbf{x}$ if $l = 1$).

This compact, vectorized form generalizes the earlier scalar backpropagation chain, allowing simultaneous computation of gradients for all weights and biases in a single layer.

To perform learning, we adjust each parameter in the direction that decreases the loss.

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(l)}} \quad , \quad \mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \cdot \frac{\partial \mathscr{L}}{\partial \mathbf{b}^{(l)}}$$

where η is the learning rate.

Batch Gradient Descent with Backpropagation

$t \leftarrow 0$, max iterations $\leftarrow 1000$
Initialize $\theta_t = \left\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\right\}_{l=1}^{L}$
Choose learning rate η

while $t <$ max iterations **do**
  Forward pass: compute all $\mathbf{a}^{(l)}, \mathbf{h}^{(l)}$ for $l = 1$ to L
  Compute output error: $\delta^{(L)} = \nabla_{\mathbf{h}^{(L)}} \mathscr{L} \odot g'(\mathbf{a}^{(L)})$

  for $l = L - 1$ to 1 **do**
    $\delta^{(l)} = \left(\mathbf{W}^{(l+1)^\top} \delta^{(l+1)}\right) \odot g'(\mathbf{a}^{(l)})$
  end for

  for $l = 1$ to L **do**
    $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \delta^{(l)} \cdot \mathbf{h}^{(l-1)^\top}$
    $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \cdot \delta^{(l)}$
  end for
  $t \leftarrow t + 1$
end while

1.4.4 Practical: MNIST Digit Classification

In this practical, we implement and train a feedforward neural network in PyTorch to classify handwritten digits from the MNIST dataset.

The MNIST dataset consists of grayscale images of digits from 0 to 9, each of size 28×28. Figure 1.17 is a sample illustration of digits from the dataset.

To begin, we import the essential PyTorch modules required for defining neural networks, optimization, dataset loading, and data transformation. The MNIST dataset is automatically downloaded in the original `.gz` compressed `ubyte` format.

We apply a sequence of transformations: first converting each image to a PyTorch tensor, and then normalizing it using the dataset's mean and standard deviation. The `DataLoader` wraps the dataset into iterable mini-batches and shuffles the training data at each epoch to improve learning.

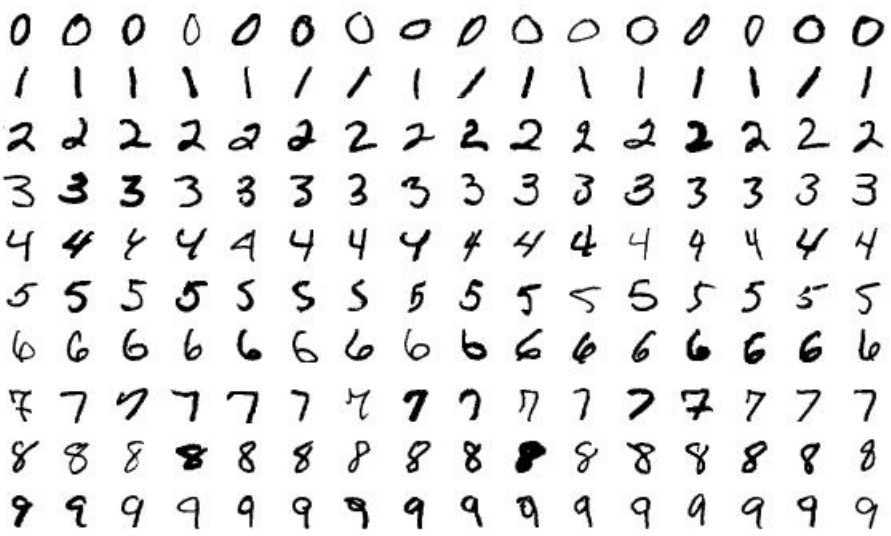


Figure 1.17: Sample images from the MNIST dataset.

```
import torch
import torch.nn as nn
import torch.optim as optim

from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, \
    transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = datasets.MNIST(root='./data', train=False, download=True, \
    transform=transform)

test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

We define a feedforward neural network by creating a subclass of `nn.Module`, which is the base class for all neural networks in PyTorch. The network structure is specified in two parts.

- The `__init__` method initializes the layers of the network. Here, the input layer is a fully connected linear layer that maps the $28 \times 28 = 784$ input pixels to 128 hidden units. This is followed by a second hidden layer with 64 units, and finally an output layer with 10 units, corresponding to the ten digit classes (0 through 9).

- The `forward` method defines the forward pass computation. The input image is first flattened to a vector of size 784. Then it is passed sequentially through the linear layers and ReLU activation functions. Specifically, the ReLU activation is applied after the first and second linear transformations to introduce non-linearity, while the final output layer produces raw scores (logits) without an activation function, typically used with `CrossEntropyLoss`.

This architecture allows the model to learn a non-linear mapping from image pixels to digit class scores using two hidden layers.

```
class FeedforwardNN(nn.Module):
    def __init__(self):
        super(FeedforwardNN, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

model = FeedforwardNN()
```

Next, we define the loss function and optimizer. We use `CrossEntropyLoss`, which internally applies `LogSoftmax` followed by negative log-likelihood loss. For optimization, we use the Adam algorithm.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

We train the model for 5 epochs. In each iteration over the training data, we first set the model to training mode using `model.train()`. For each mini-batch, we clear the accumulated gradients from the previous step using `optimizer.zero_grad()`, perform a forward pass to compute the predictions, and then compute the loss using the specified loss function.

The backpropagation step is triggered by `loss.backward()`, which computes the gradients of the loss with respect to the model parameters. These gradients are then used by the optimizer to update the model parameters via `optimizer.step()`. A log of the loss is printed every 100 batches to monitor training progress.

```
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], \
                Batch [{batch_idx}/{len(train_loader)}], Loss: {loss.item():.4f}")
```

After training, we evaluate the model's accuracy on the test set. We disable gradient computation and compare predicted labels with true labels to count the correct predictions.

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f'Test Accuracy of the model on 10,000 test images: \
    {100 * correct / total:.2f}%')
```

The trained model achieves an accuracy of approximately **97.65%** on the MNIST test dataset.

This confirms that even a basic feedforward neural network, when trained appropriately using backpropagation, can effectively learn to classify digits from images.

Review: Assignment 1

[1]

The objective of this assignment is to implement and use gradient descent (and its variants) with backpropagation for a classification task using a feedforward neural network. The task is to classify images from the Fashion-MNIST dataset (available on Kaggle) into one of 10 classes.

You are expected to implement the full network from scratch using Python. You may use `numpy` and `pandas`, but not any external deep learning libraries like PyTorch or TensorFlow. Use of the `argparse` module for command-line interface is mandatory. Set the seed to 1234 using `numpy.random.seed(1234)` to ensure replicability.

Your implementation must support the following command-line options.

```
--lr          : initial learning rate
--momentum    : momentum term (used only in momentum-based methods)
--num_hidden  : number of hidden layers
--sizes       : comma-separated list of sizes for each hidden layer
--activation  : activation function (tanh or sigmoid)
--loss        : loss function (sq for squared error, ce for cross entropy)
--opt         : optimization method (gd, momentum, nag, adam)
--batch_size  : batch size (1 or a multiple of 5)
--anneal      : halve learning rate if validation loss decreases (true/false)
--save_dir    : directory to save the model (weights and biases)
--expt_dir    : directory to save logs and predictions
--train       : path to training dataset
--test        : path to test dataset
```

Your main script must be named `train.py` and should run using the following format.

```
python train.py --lr 0.01 --momentum 0.5 --num_hidden 3 --sizes 100,100,100 \
--activation sigmoid --loss sq --opt adam --batch_size 20 --anneal true \
--save_dir pa1/ --expt_dir pa1/exp1/ --train train.csv --test test.csv
```

You must log your training and validation metrics every 100 steps in the files `log_train.txt` and `log_val.txt`, located in the `expt_dir`.

Each line must be formatted as follows.

```
Epoch 0, Step 100, Loss: <value>, Error: <value>, lr: <value>
```

The error is the percentage of incorrect predictions (rounded to two decimal places).

You must also generate a file `predictions.csv` in the `expt_dir`, containing the test set predictions with the following format.

```
id,label
0,3
1,2
2,8
...
9999,4
```

Your report must be written in LaTeX and include the following experimental plots.

1. Varying hidden layer sizes
For one, two, three, and four hidden layers, each with sizes in {50, 100, 200, 300}, plot the training and validation loss vs epoch (4 curves per plot, one for each size). Use sigmoid activation, cross entropy loss, Adam optimizer, batch size 20, and tune learning rate.

2. Varying optimization algorithms
Use 3 hidden layers of size 300 each and plot the training and validation loss using GD, momentum, NAG, and Adam.

3. Activation functions comparison
Compare sigmoid and tanh activations with 2 hidden layers of size 100, using Adam, cross entropy loss, and batch size 20.

4. Loss functions comparison
Compare squared error and cross entropy loss with 2 hidden layers of size 100, using sigmoid activation, Adam, and batch size 20.

5. Batch size comparison
Test with batch sizes 1, 20, 100, and 1000, using 2 hidden layers of size 100, sigmoid activation, cross entropy loss, and Adam.

Each plot must have epochs on the x-axis and loss on the y-axis, with clear legends.

You must create a file `supported.txt` listing the supported options for `-anneal`, `-opt`, `-loss`, and `-activation`, e.g.,

```
--anneal: true,false
--opt: gd,momentum,nag,adam
--loss: sq,ce
--activation: tanh,sigmoid
```

All deliverables should be packaged into a single `tar.gz` archive named `RollNo_backprop.tar.gz`, containing

```
- train.py
- run.sh (best performing command)
- any other Python scripts
- supported.txt
- report.pdf (written in LaTeX)
- predictions.csv
```

Your task is to achieve an error rate below 8% on the test set. Evaluation will be based on performance, code correctness, completeness, and the ability to support the specified hyperparameters.

□

Chapter 2

Low-Rank Representations & Autoencoding

Most of modern machine learning operates on vectors. Regardless of the nature of input — whether it is numerical data, images, audio signals, or natural language — the first step is to convert it into a numerical format that a computer can process: a vector in $\mathbb{R}^d$. This is not merely a convenience; it is a necessity. Computers are optimized to perform linear algebra and numerical computation at scale, and vectors form the basis of such operations. From computing distances and similarities to matrix multiplications and optimization, everything depends on representing data in vector form.

Let's see how various data modalities are turned into vectors.

- **Numeric Data**: Structured datasets with continuous or categorical variables like age, height, salary, etc., are naturally vectorized. For example, the features $\{\text{height} = 170 \text{ cm}, \text{weight} = 65 \text{ kg}, \text{age} = 29 \text{ yrs}\}$ can be represented as $\mathbf{x} = [170, 65, 29]^\top$. In practice, these vectors are often normalized to have zero mean and unit variance to help the optimization process.

- **Images**: An image is a grid of pixels, each with intensity values. A grayscale image of size 28×28 (like MNIST) has 784 pixels and can be flattened into a vector in $\mathbb{R}^{784}$. Colored images contain multiple channels, typically three for RGB. So an image of size 32×32 with 3 channels becomes a vector in $\mathbb{R}^{3072}$. In neural networks, this flattening is often deferred in case of images until after several layers, but conceptually, the image is still a point in a high-dimensional space.

- **Audio**: A raw audio signal is a 1D waveform sampled at a fixed rate. A mono audio of 1 second at a 16kHz sampling rate results in a vector of $16,000$ samples in $\mathbb{R}^{16000}$. Audio is often chunked into frames and processed using spectrograms or Mel-frequency cepstral coefficients (MFCCs), which are again vector representations derived from the original waveform.

- **Text**: Text data is symbolic and lacks a native numeric representation. To embed text in vector spaces, we use encoding schemes. At the simplest level, we can use one-hot encoding. But more powerful representations include dense word embeddings (like Word2Vec, GloVe) where each word maps to a dense vector in $\mathbb{R}^d$, often with d ranging from 50 to 300. We will see this in detail in Section 2 of this book.

Once each input modality is converted into a vector, we can begin applying machine learning models. However, a common challenge emerges: the ***curse of dimensionality***. As the number of dimensions increases, the volume of the space grows exponentially, and data points become sparse. This sparsity can severely impair learning algorithms. Intuitively, when data lives in a very high-dimensional space, it becomes difficult to learn generalizable patterns unless we have an enormous amount of training data.

For example, if each input vector has $10,000$ dimensions, and we have only 100 training samples, then most standard learning algorithms will overfit. They will memorize the training data but fail to generalize to unseen examples. This phenomenon is not just theoretical; it has real-world consequences. The model may show perfect accuracy during training but perform poorly during deployment.

Low-rank representations offer a solution to this problem. The idea is to reduce the dimensionality of the data while retaining most of the relevant information. This is done by projecting high-dimensional vectors onto a lower-dimensional subspace. Mathematically, we seek a low-rank approximation of the data matrix $X \in \mathbb{R}^{n \times d}$, where n is the number of samples and d is the original dimensionality. Techniques like Principal Component Analysis (PCA), autoencoders, or random projections are used to obtain these reduced representations.

These compressed vectors are often referred to as ***latent representations***. They are called *latent* because they are not observed directly but are inferred from the data. In deep learning, especially, the intermediate hidden layers of a neural network can be interpreted as learning such latent spaces. These representations often disentangle the factors of variation in the data. So, low-rank representations are not just about compression or storage efficiency.

2.1 Principal Component Analysis

Principal Component Analysis (PCA) is a method for reducing the number of variables in a dataset while keeping as much information as possible. It works by finding new directions, called *principal components*, that capture the most variation in the data. These directions are linear combinations of the original variables.

2.1.1 Interpretation 1: High Variance In New Basis

One way to understand PCA is to think about variance. PCA looks for directions in which the data varies the most. *Why do we care about variance?* Because variance tells us where the information in the data lies. If a direction has high variance, it means that the data points are spread out along that direction. So, when we project the data onto such directions, we retain more of the structure and differences among data points.

Suppose we have a centered data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where each row $\mathbf{x}_i$ is a data point and each column has zero mean. We want to find a unit vector $\mathbf{w} \in \mathbb{R}^d$ that maximizes the variance of the data when projected onto $\mathbf{w}$.

The projection of data points onto $\mathbf{w}$ is given by

$$\mathbf{X}\mathbf{w} = \begin{bmatrix} \mathbf{x}_1^\top \mathbf{w} \\ \mathbf{x}_2^\top \mathbf{w} \\ \vdots \\ \mathbf{x}_n^\top \mathbf{w} \end{bmatrix} \in \mathbb{R}^n.$$

The variance of these projected values is

$$\text{Var}(\mathbf{X}\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i^\top \mathbf{w})^2 = \frac{1}{n}\|\mathbf{X}\mathbf{w}\|^2 = \mathbf{w}^\top \left(\frac{1}{n}\mathbf{X}^\top \mathbf{X}\right)\mathbf{w} = \mathbf{w}^\top \Sigma \mathbf{w},$$

where $\Sigma = \frac{1}{n}\mathbf{X}^\top \mathbf{X}$ is the covariance matrix.

Our goal is to find $\mathbf{w}$ that maximizes this variance, with the constraint that $\mathbf{w}$ is a unit vector.

$$\max_{\mathbf{w}\in\mathbb{R}^d} \mathbf{w}^\top \Sigma \mathbf{w} \quad \text{subject to} \quad \mathbf{w}^\top \mathbf{w} = 1.$$

We solve this constrained optimization using the method of Lagrange multipliers. We define the Lagrangian

$$\mathscr{L}(\mathbf{w}, \lambda) = \mathbf{w}^\top \Sigma \mathbf{w} - \lambda(\mathbf{w}^\top \mathbf{w} - 1),$$

where λ is the Lagrange multiplier. To find stationary points, take the gradient with respect to $\mathbf{w}$ and set it to zero.

$$\nabla_{\mathbf{w}} \mathscr{L} = 2\Sigma\mathbf{w} - 2\lambda\mathbf{w} = \mathbf{0} \implies \Sigma\mathbf{w} = \lambda\mathbf{w}.$$

This is the standard eigenvalue equation. The vector $\mathbf{w}$ must be an eigenvector of Σ, and λ the corresponding eigenvalue.

Recall we want to maximize $\mathbf{w}^\top \Sigma \mathbf{w}$. Using the eigenvector property, $\mathbf{w}^\top \Sigma \mathbf{w} = \mathbf{w}^\top(\lambda\mathbf{w}) = \lambda\mathbf{w}^\top\mathbf{w} = \lambda$, since $\mathbf{w}$ is unit norm.

So, the variance equals the eigenvalue λ. To maximize variance, choose $\mathbf{w}$ as the eigenvector corresponding to the largest eigenvalue λ_1. This $\mathbf{w}_1$ is the *first principal component*. The variance along $\mathbf{w}_1$ is λ_1.

To find the second principal component $\mathbf{w}_2$, we maximize variance with the constraint that $\mathbf{w}_2$ is orthogonal to $\mathbf{w}_1$.

$$\max_{\mathbf{w}_2} \mathbf{w}_2^\top \Sigma \mathbf{w}_2, \quad \text{subject to } \mathbf{w}_2^\top \mathbf{w}_2 = 1, \quad \mathbf{w}_2^\top \mathbf{w}_1 = 0.$$

The solution is the eigenvector corresponding to the second largest eigenvalue λ_2, and so on.

2.1.2 Interpretation 2: Minimum Covariance In New Basis

One way to understand PCA is through its effect on the covariance structure of the data. In the original space, features may be correlated, which can make the data harder to interpret or model. PCA changes the coordinate system by rotating the data onto a new set of orthogonal axes, *principal components*, where these correlations are removed.

Let $\mathbf{X} \in \mathbb{R}^{n\times d}$ be the zero-mean data matrix. The sample covariance matrix of the original data is given by

$$\Sigma = \frac{1}{n}\mathbf{X}^\top \mathbf{X}$$

Let $\mathbf{W} \in \mathbb{R}^{d\times k}$ be the matrix whose columns are the top k eigenvectors of Σ. The projected data in the new basis is

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

Then the covariance matrix of the projected data is

$$\Sigma_{\mathbf{Z}} = \frac{1}{n}\mathbf{Z}^\top \mathbf{Z} = \frac{1}{n}\mathbf{W}^\top \mathbf{X}^\top \mathbf{X}\mathbf{W} = \mathbf{W}^\top \Sigma \mathbf{W}$$

Since the columns of $\mathbf{W}$ are the eigenvectors of Σ, the expression $\mathbf{W}^\top \Sigma \mathbf{W}$ becomes a diagonal matrix

$$\Sigma_{\mathbf{Z}} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_k \end{bmatrix}$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$ are the top k eigenvalues of Σ. This means that in the PCA-transformed space, the features (i.e., the principal components) are uncorrelated, and each one captures a specific portion of the total variance in the data.

This interpretation shows that PCA not only reduces dimensionality but also simplifies the covariance structure. In the new basis, the data has no cross-correlation between dimensions, making downstream tasks like regression or classification more stable and interpretable. It essentially decorrelates the variables by design.

2.1.3 Interpretation 3: Best Low-Rank Representation

The most rigorous interpretation of PCA is that it gives the best low-rank approximation to the data in terms of minimizing reconstruction error. This means PCA finds a lower-dimensional subspace that captures as much of the original data as possible, while reducing dimensionality in a way that the reconstruction from this subspace is as close as possible to the original.

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the data matrix, where each row is a centered data point (i.e., the column means of $\mathbf{X}$ are zero). We aim to find a rank-k approximation $\hat{\mathbf{X}}$ of $\mathbf{X}$, with $k < d$, such that the reconstruction error is minimized. That is,

$$\min_{\text{rank}(\hat{\mathbf{X}})=k} \|\mathbf{X} - \hat{\mathbf{X}}\|_F^2$$

where $\|\cdot\|_F$ denotes the Frobenius norm, which sums the squared entries of the matrix.

We project the data onto a k-dimensional subspace using a projection matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$, whose columns are orthonormal vectors, i.e., $\mathbf{W}^\top \mathbf{W} = \mathbf{I}_k$. The data projected into this subspace is $\mathbf{Z} = \mathbf{X}\mathbf{W}$. We can then reconstruct the original data from this lower-dimensional representation as

$$\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top = \mathbf{X}\mathbf{W}\mathbf{W}^\top$$

The reconstruction error is

$$\|\mathbf{X} - \hat{\mathbf{X}}\|_F^2 = \|\mathbf{X} - \mathbf{X}\mathbf{W}\mathbf{W}^\top\|_F^2 = \text{Tr}[(\mathbf{X} - \mathbf{X}\mathbf{W}\mathbf{W}^\top)^\top (\mathbf{X} - \mathbf{X}\mathbf{W}\mathbf{W}^\top)]$$

$$= \text{Tr}(\mathbf{X}^\top \mathbf{X}) - \text{Tr}(\mathbf{X}^\top \mathbf{X}\mathbf{W}\mathbf{W}^\top) - \text{Tr}(\mathbf{W}\mathbf{W}^\top \mathbf{X}^\top \mathbf{X}) + \text{Tr}(\mathbf{W}\mathbf{W}^\top \mathbf{X}^\top \mathbf{X}\mathbf{W}\mathbf{W}^\top)$$

Using cyclic properties of the trace and the fact that $\mathbf{W}^\top \mathbf{W} = \mathbf{I}_k$, this simplifies to

$$= \text{Tr}(\mathbf{X}^\top \mathbf{X}) - \text{Tr}(\mathbf{W}^\top \mathbf{X}^\top \mathbf{X}\mathbf{W})$$

Thus, minimizing the reconstruction error is equivalent to maximizing the trace

$$\max_{\mathbf{W}^\top \mathbf{W} = \mathbf{I}_k} \operatorname{Tr}(\mathbf{W}^\top \mathbf{X}^\top \mathbf{X} \mathbf{W})$$

The matrix $\mathbf{X}^\top \mathbf{X}$ is the unnormalized covariance matrix of the data (scaled by n). This is a symmetric, positive semi-definite matrix. The solution to this optimization problem is given by setting the columns of $\mathbf{W}$ to be the top k eigenvectors of $\mathbf{X}^\top \mathbf{X}$ corresponding to the top k eigenvalues.

Therefore, PCA finds a new basis of k orthogonal directions (eigenvectors) that capture the largest variance in the data and provide the best low-rank reconstruction in the least-squares sense.

2.2 Singular Value Decomposition

So far, we've talked about square matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$. But *what if the matrix is rectangular, say* $\mathbf{A} \in \mathbb{R}^{m \times n}$*? Can it still have eigenvectors?*

Suppose we try to apply $\mathbf{A}$ on a vector $\mathbf{x} \in \mathbb{R}^n$. The result $\mathbf{A}\mathbf{x}$ belongs to $\mathbb{R}^m$, not $\mathbb{R}^n$. So the equation $\mathbf{A}\mathbf{x} = \lambda \mathbf{x}$ doesn't make sense anymore because both sides aren't in the same space. This means rectangular matrices can't have eigenvectors in the same way square ones do.

However, $\mathbf{A}$ still defines a transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$. What if we could find pairs of vectors $(\mathbf{v}_i, \mathbf{u}_i)$ such that

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$$

where $\mathbf{v}_i \in \mathbb{R}^n$, $\mathbf{u}_i \in \mathbb{R}^m$, and σ_i are scalars. If the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ form an orthogonal basis for a subspace of $\mathbb{R}^n$, and $\mathbf{u}_1, \ldots, \mathbf{u}_k$ form an orthogonal basis for a subspace of $\mathbb{R}^m$, then any $\mathbf{x} \in \mathbb{R}^n$ can be expressed as

$$\mathbf{x} = \sum_{i=1}^{k} \alpha_i \mathbf{v}_i$$

Then applying $\mathbf{A}$ becomes

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{k} \alpha_i \mathbf{A}\mathbf{v}_i = \sum_{i=1}^{k} \alpha_i \sigma_i \mathbf{u}_i$$

So once again, the matrix multiplication simplifies into scalar multiplications. Here, $k = \operatorname{rank}(\mathbf{A})$.

We can write the equations

$$\mathbf{A}\mathbf{v}_1 = \sigma_1 \mathbf{u}_1, \quad \mathbf{A}\mathbf{v}_2 = \sigma_2 \mathbf{u}_2, \quad \ldots, \quad \mathbf{A}\mathbf{v}_k = \sigma_k \mathbf{u}_k$$

or compactly,

$$\mathbf{A}\mathbf{V} = \mathbf{U}\Sigma$$

where $\mathbf{V} \in \mathbb{R}^{n \times k}$ contains the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ as columns, $\mathbf{U} \in \mathbb{R}^{m \times k}$ contains the vectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ as columns, and $\Sigma \in \mathbb{R}^{k \times k}$ is a diagonal matrix with entries $\sigma_1, \ldots, \sigma_k$.

We can extend the orthogonal vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ to a full orthogonal basis of $\mathbb{R}^n$ using Gram-Schmidt. Similarly, we can complete $\mathbf{u}_1, \ldots, \mathbf{u}_k$ to a full orthogonal basis of $\mathbb{R}^m$. With this, we get the full SVD.

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$$

Here $\mathbf{U} \in \mathbb{R}^{m \times m}$ is an orthogonal matrix (left singular vectors), $\mathbf{V} \in \mathbb{R}^{n \times n}$ is an orthogonal matrix (right singular vectors), and $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k > 0$ on the diagonal.

To find $\mathbf{U}$, $\mathbf{V}$, and Σ, assume the decomposition exists

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$$

Then

$$\mathbf{A}^\top\mathbf{A} = (\mathbf{U}\Sigma\mathbf{V}^\top)^\top(\mathbf{U}\Sigma\mathbf{V}^\top) = \mathbf{V}\Sigma^\top\mathbf{U}^\top\mathbf{U}\Sigma\mathbf{V}^\top = \mathbf{V}\Sigma^2\mathbf{V}^\top$$

So $\mathbf{A}^\top\mathbf{A}$ has the same eigenvectors as $\mathbf{V}$ and eigenvalues Σ^2. Similarly,

$$\mathbf{A}\mathbf{A}^\top = \mathbf{U}\Sigma\mathbf{V}^\top\mathbf{V}\Sigma^\top\mathbf{U}^\top = \mathbf{U}\Sigma^2\mathbf{U}^\top$$

Thus, $\mathbf{V}$ contains the eigenvectors of $\mathbf{A}^\top\mathbf{A}$, $\mathbf{U}$ contains the eigenvectors of $\mathbf{A}\mathbf{A}^\top$, and the non-zero eigenvalues of both are the same.

$$\sigma_i = \sqrt{\lambda_i} \quad \text{(singular values)}$$

Symbol	Meaning
σ_i	Singular value of $\mathbf{A}$
$\mathbf{U}$	Left singular matrix (eigenvectors of $\mathbf{A}\mathbf{A}^\top$)
$\mathbf{V}$	Right singular matrix (eigenvectors of $\mathbf{A}^\top\mathbf{A}$)
Σ	Diagonal matrix with singular values

Table 2.1: SVD Components and Interpretations

2.3 Autoencoders and PCA

An autoencoder is a type of feedforward neural network with two parts: an encoder and a decoder.

The encoder maps the input $\mathbf{x}_i \in \mathbb{R}^d$ to a hidden representation $\mathbf{h} \in \mathbb{R}^k$, where typically $k < d$. This hidden representation $\mathbf{h}$ is often called the *code* or the *bottleneck*.

$$\mathbf{h} = f_{\text{enc}}(\mathbf{x}_i)$$

The decoder tries to reconstruct the input from the hidden representation.

$$\hat{\mathbf{x}}_i = f_{\text{dec}}(\mathbf{h})$$

The model is trained to minimize the reconstruction loss between $\mathbf{x}_i$ and $\hat{\mathbf{x}}_i$, often using the squared error loss.

$$\mathscr{L} = \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2$$

Now, suppose the dimension of the hidden representation is smaller than that of the input, i.e., $\dim(\mathbf{h}) < \dim(\mathbf{x}_i)$. This forces the network to compress the input information. If the decoder can still reconstruct $\mathbf{x}_i$ perfectly from $\mathbf{h}$, it tells us something important: the hidden representation $\mathbf{h}$ has preserved all the necessary information from $\mathbf{x}_i$. *Do you see an analogy with PCA?*

2.3.1 Link Between PCA and Autoencoders

We now show that the encoder in an autoencoder is equivalent to Principal Component Analysis (PCA) under the following conditions.

- The encoder is linear
- The decoder is linear
- The loss function is squared error
- The input data is normalized as

$$\hat{x}_{ij} = \frac{1}{\sqrt{m}} \left(x_{ij} - \frac{1}{m} \sum_{k=1}^{m} x_{kj} \right)$$

Let us examine the effect of this normalization. Define $\hat{\mathbf{X}}$ to be the normalized input matrix. The expression inside the parentheses centers the data along each feature j by subtracting the mean. Let $\mathbf{X}^0$ denote the zero-mean data matrix.

$$\hat{\mathbf{X}} = \frac{1}{\sqrt{m}} \mathbf{X}^0 \implies \hat{\mathbf{X}}^\top \hat{\mathbf{X}} = \frac{1}{m} (\mathbf{X}^0)^\top \mathbf{X}^0$$

This is the covariance matrix of the centered data, which plays a central role in PCA.

Now we minimize the squared error loss using a linear encoder and decoder.

$$\min_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{n} (x_{ij} - \hat{x}_{ij})^2$$

In matrix notation, this is

$$\min_{\mathbf{W}^*, \mathbf{H}} \|\mathbf{X} - \mathbf{H}\mathbf{W}^*\|_F^2$$

where $\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$ is the Frobenius norm.

From the Singular Value Decomposition (SVD), the optimal solution to this is

$$\mathbf{H}\mathbf{W}^* = \mathbf{U}_{:,\leq k} \mathbf{\Sigma}_{k,k} \mathbf{V}_{:,\leq k}^\top$$

One possible solution (by matching terms) is

$$\mathbf{H} = \mathbf{U}_{:,\leq k} \mathbf{\Sigma}_{k,k}, \quad \mathbf{W}^* = \mathbf{V}_{:,\leq k}^\top$$

We now derive the encoder weights and verify that the encoder is linear.

$$
\begin{aligned}
\mathbf{H} &= \mathbf{U}_{:,\leq k}\Sigma_{k,k} \\
&= (\mathbf{X}\mathbf{X}^\top)(\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{U}_{:,\leq k}\Sigma_{k,k} \\
&= (\mathbf{X}\mathbf{V}\Sigma^\top\mathbf{U}^\top)(\mathbf{U}\Sigma\Sigma^\top\mathbf{U}^\top)^{-1}\mathbf{U}_{:,\leq k}\Sigma_{k,k} \quad (\text{since } \mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top) \\
&= \mathbf{X}\mathbf{V}\Sigma^\top\mathbf{U}^\top(\mathbf{U}\Sigma\Sigma^\top\mathbf{U}^\top)^{-1}\mathbf{U}_{:,\leq k}\Sigma_{k,k} \\
&= \mathbf{X}\mathbf{V}\Sigma^\top(\Sigma\Sigma^\top)^{-1}\mathbf{U}^\top\mathbf{U}_{:,\leq k}\Sigma_{k,k} \\
&= \mathbf{X}\mathbf{V}\Sigma^\top\Sigma^{-1}\Sigma^{-1}\mathbf{U}^\top\mathbf{U}_{:,\leq k}\Sigma_{k,k} \\
&= \mathbf{X}\mathbf{V}\Sigma^{-1}\mathbf{I}_{:,\leq k}\Sigma_{k,k} \quad (\text{since } \mathbf{U}^\top\mathbf{U}_{:,\leq k} = \mathbf{I}_{:,\leq k}) \\
\mathbf{H} &= \mathbf{X}\mathbf{V}_{:,\leq k}
\end{aligned}
$$

Thus, $\mathbf{H}$ is a linear transformation of $\mathbf{X}$, and the encoder matrix is

$$\mathbf{W} = \mathbf{V}_{:,\leq k}$$

From SVD, $\mathbf{V}$ contains the eigenvectors of $\mathbf{X}^\top\mathbf{X}$. From PCA, the projection matrix $\mathbf{P}$ also consists of the eigenvectors of the covariance matrix.

If we normalize $\mathbf{X}$ as

$$\hat{x}_{ij} = \frac{1}{\sqrt{m}}\left(x_{ij} - \frac{1}{m}\sum_{k=1}^{m} x_{kj}\right)$$

then $\mathbf{X}^\top\mathbf{X}$ becomes the covariance matrix.

Hence, the encoder weights from the linear autoencoder and the PCA projection matrix are the same.

2.4 Regularization in Autoencoders

Regularization in autoencoders helps prevent overfitting and encourages the model to learn meaningful features. Instead of just copying the input to the output, regularization forces the model to generalize. Below are three common ways this is done.

2.4.1 Denoising Autoencoders

Denoising autoencoders add noise to the input data, then train the model to reconstruct the original input from the noisy version. The idea is that the network must learn the underlying structure of the data to perform this task well.

Let $\tilde{\mathbf{x}}$ be the noisy input generated from the original input $\mathbf{x}$ using a corruption process (e.g., Gaussian noise, masking noise). The encoder learns a hidden representation $\mathbf{h} = f(\tilde{\mathbf{x}})$, and the decoder reconstructs $\hat{\mathbf{x}} = g(\mathbf{h})$. The loss is computed between $\mathbf{x}$ and $\hat{\mathbf{x}}$, not between $\tilde{\mathbf{x}}$ and $\hat{\mathbf{x}}$.

This regularization discourages the model from simply memorizing the input.

2.4.2 Sparse Autoencoders

Sparse autoencoders apply a sparsity constraint on the hidden layer activations. The idea is to force most of the hidden units to be inactive (close to zero) for a given input. This leads to learning a set of features where only a few are active at a time, making the representation more efficient and interpretable.

This is often done by adding a penalty term to the loss function. Let ρ be the desired sparsity level (e.g., 0.05), and $\hat{\rho}_j$ be the average activation of hidden unit j over the training set. A common choice for the sparsity penalty is the KL-divergence

$$\sum_{j=1}^{n_{\text{hidden}}} \text{KL}(\rho \| \hat{\rho}_j) = \sum_{j=1}^{n_{\text{hidden}}} \left(\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \right)$$

The overall loss becomes

$$\mathscr{L} = \text{Reconstruction Loss} + \beta \sum_j \text{KL}(\rho \| \hat{\rho}_j)$$

where β controls the strength of the sparsity constraint.

2.4.3 Contractive Autoencoders

Contractive autoencoders penalize the sensitivity of the encoder to small changes in the input. This is done by adding the Frobenius norm of the Jacobian of the hidden representation with respect to the input to the loss function.

Let $\mathbf{h} = f(\mathbf{x})$. The regularization term is

$$\lambda \left\| \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \right\|_F^2$$

This term encourages the hidden representation to be locally invariant to small changes in input. As a result, the model learns more robust and stable features. The full loss is

$$\mathscr{L} = \text{Reconstruction Loss} + \lambda \left\| \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \right\|_F^2$$

Chapter 3

Better Model Training Techniques

Out of two models, which one is better? And what are the tips and techniques that we shall follow in order to make a better model?

3.1 Bias and Variance

3.1.1 The Tradeoff

3.1.2 Train Error vs. Test Error

3.1.3 True Error and Model Complexity

3.2 Regularization

3.2.1 l_2 Regularization

3.2.2 Dataset Augmentation

3.2.3 Parameter Sharing and Tying

3.2.4 Adding Noise to the Inputs

3.2.5 Adding Noise to the Outputs

3.2.6 Early Stopping

3.2.7 Ensemble Methods

3.2.8 Dropout

3.3 Better Activation Functions

3.4 Better Weight Initialization

3.5 Batch Normalization

SECTION 2

LANGUAGE AND TEXT

Chapter 4

Introduction to Natural Language Processing

Since computers can only understand the numbers, the question was: how to teach computers about the language?

Why? What's the motivation for this? What tasks do we want to do?

4.1 N-Gram Language Model

4.2 Naive Bayes' Model

4.3 Logistic Regression for Text Classification

Chapter 5

Word Embeddings

Some intro

5.1 Word2Vec Model

5.2 Skip Gram Model

5.3 Feedforward Neural Language Model

5.4 Recurrent Neural Network

5.5 Long Short Term Memory Cells

Chapter 6

The Transformer

Some intro

6.1 Attention

6.2 Transformer Blocks

6.3 Computational Enhancements

6.4 Transformer Language Model

Chapter 7

Large Language Models

Some intro

7.1 Transformer Architecture

7.2 Sampling for LLMs

7.3 Pretraining LLMs

7.4 Evaluating LLMs

7.5 Scaling LLMs

7.6 Problems with LLMs

Chapter 8

Masked Language Models

Some intro

8.1 Bidirectional Transformer Encoders

8.2 Training Bidirectional Encoders

8.3 Contextual Embeddings

8.4 Fine-Tuning for Classification

Chapter 9

Prompting and Model Alignment

Some intro

9.1 Prompting

9.2 Post-Training and Model Alignment

9.3 Prompt Engineering

Chapter 10

Retrieval Augmented Generation

Chapter 11

LangChain and LangSmith

Chapter 12

LangGraph and AI Agents

Chapter 13

Ollama Ecosystem

SECTION 3

VISION AND IMAGES

SECTION 4

AUDIO AND SPEECH

SECTION 5

VIDEO, EMBODIMENT, AND INTERACTION

SECTION 6

SYSTEMS, ETHICS, AND THE FUTURE

Bibliography

[1] Mitesh M. Khapra. Cs7015: Deep learning. https://www.cse.iitm.ac.in/~miteshk/CS7015_2018.html, 2018. Lecture Series, Department of Computer Science and Engineering, IIT Madras.

[2] Warren S. McCulloch; Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, Society for Mathematical Biology*, 5, 1943.

[3] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.